TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

MIIKKA JUOMOJA

TRANSPARENT DISPLAY'S GRAPHICAL UNIT

Master's thesis

Examiner: professor Hannu-Matti Järvinen
Examiner and topic approved at Computing and Electrical engineering Academic Board 4 May 2016

ABSTRACT

MIIKKA JUOMOJA: Transparent display's graphical unit
Tampere University of Technology
Master of Science Thesis, 45 pages, 25 Appendix pages
May 2016
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Professor Hannu-Matti Järvinen

Keywords: Linux driver, electroluminescent display, graphical engine, transparency, SPI

This master's thesis is about graphical logic unit and supporting device driver which are part of a project where four transparent displays are embedded into vehicle's windshield. The project is started by Pilkington Automotive Oy which specialises into windshield production. The company uses Beneq Product Oy's transparent Lumineq displays inside their windshield prototypes. The Tampere University of Technology is part of the project producing all the necessary electronics and software for the prototype. This thesis focuses on the Linux SPI device driver and the graphical aspect of the software logic.

The software listens the vehicle's CAN bus and optionally user-based messages through the TCP/IP protocol. The is a control unit which task is to decide which actions are executed. The control unit uses the graphical engine's API to change the displays into their new desired state. The graphics engine utilises custom messages which are transformed into Beneq's custom low level SPI messages that control the electroluminescent displays. The master's thesis focuses on the architectures, communication and methods used in the graphical engine.

The software is built to be as modular as possible. After the prototype stage the end clients decide how the displays are built and what items they include. The architecture is decided to be built fully customisable through YAML configuration files. The graphical engine consists of two parts: The first part handles the parsing of the YAML configuration files at the beginning of the software. The second part reads incoming messages from the control unit, executes necessary tests, creates timing if needed and passes the parsed commands into the Linux device driver.

TIIVISTELMÄ

MIIKKA JUOMOJA: Läpinäkyvän näytön graafinen ohjelmisto
Tampereen teknillinen yliopisto
Diplomityö, 45 sivua, 25 liitesivua
Toukokuu 2016
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Hannu-Matti Järvinen

Avainsanat: Linux ajuri, elektroluminenssinäyttö, graafinen moottori, läpinäkyvyys, SPI

Tässä diplomityössä toteutettiin graafisen logiikan ohjelmisto sekä ohjelmistoajuri osana tuulilasin sisään upotetun läpinäkyvän näyttökokonaisuuden projektia. Projektin on aloittanut tuulilaseihin erikoistunut Pilkington Automotive Oy, joka hyödyntää Beneq Products Oy:n läpinäkyviä Lumineq näyttöjä prototyyppilasissaan. Tampereen teknillisen yliopiston tehtävänä oli tuottaa prototyyppiin tarvittavat elektroniikat sekä ohjelmisto. Tämä diplomityö on rajattu käsittelemään ohjelmistopuolelta Linux SPI -ajuria sekä ohjelmiston graafista logiikkaa koskevaa osaa.

Kokonaisuutena ohjelmisto kuuntelee sekä ajoneuvon CAN-väylää että mahdollisia käyttöliittymiä TCP/IP-yhteyden avulla. Looginen puoli käsittelee viestit ja välittää sen päättelemät tehtävät graafiselle ohjelmistolle, joka muokkaa käskyt näytöille sopiviksi. Käskyt lähetetään SPI-ajurimoduulin kautta suoraan näytöille. Työssä tarkastellaan ohjelmiston arkkitehtuureja ja toteutustapoja.

Ohjelmistosta päätettiin rakentaa mahdollisimman muokattava, sillä prototyypin jälkeen asiakkaat päättävät, miten näytöt rakennetaan ja mitä ne sisältävät. Tämän vuoksi arkkitehtuurissa päädyttiin rakentamaan täysin muokattava ohjelmisto YAML-konfiguraatiotiedostojen avulla. Rakenteessa graafinen puoli koostuu kahdesta osasta: ensimmäinen lukee alussa käyttäjän luomat konfiguraatiot, joita toinen osa hyödyntää omassa graafisessa logiikassaan. Toinen osa kuuntelee sisään tulevia käskyjä, suorittaa tarvittavat tarkistustoimenpiteet, luo ajastuksen tarvittaessa ja välittää tiedot ajurille.

PREFACE

This thesis is made for the department of pervasive computing at Tampere University of Technology. The work is a part of a prototype project which is made for Pilkington Automotive Oy. The examiner was professor Hannu-Matti Järvinen, which I would like to thank for the inspection of this thesis and from excellent guidance. I would also thank Tommi Rekosuo from excellent cooperation with this project and the staff of TUT's department of pervasive computing. I am dedicating this master's thesis for my wife Liisa who has supported me through the studies.

Tampere 4.5.2016

Miikka Juomoja

TABLE OF CONTENTS

LIST OF FIGURES

ACRONYMS AND GLOSSARY

ACK	Message acknowledged
C	Programming language
CALLOC	Memory allocation function at Standard C Libraries
CAN	Bus used mostly at vehicles
Control unit	Software which holds the communication and the logic
CPHA	SPI bus clock phase
CPOL	SPI bus clock polarity
CS	Chip select
Display Unit	The software which holds the display logic of the project
Ethernet	Computer networking technologies
FIFO	First-in, first-out named pipe
Frame rate	Frame frequency
Fuzz test	Test where random or invalid data is pushed into a software
GDB	The GNU Debugger
GPIO	General-purpose input/output pin
GPL	The GNU General Public License
Group	A group of segments which cannot be used in navigation
Icon	A group of segments which can be used in navigation
IOCTL	Input/output control
JSON	JavaScript Object Notation
LCD	Liquid-crystal display
LibYAML	YAML parser library for C programming language
Linux	Operating system
Linux Driver Module	Program which controls hardware device
MISO	Master Input, Slave Output
MOSI	Master Output, Slave Input
NACK	Message not acknowledged
Peer review	Meeting where a set of code is reviewed and commented
PyYAML	Python YAML parser library which uses LibYAML as base
QT	Cross-platform application framework
SD	Secure Digital memory card
Segment	Part of a display which can be set on or off
Performance test	Test which test the speed and the burden of the program
SPI	Serial Peripheral Interface Bus
SPIDEV	SPI devices API
Splint	Static code analysis software
SS	Slave Select
STRCMP	String compare function at standard C libraries
STRLEN	String length function at standard C libraries
SVN	Apache Subversion, software revisioning and version control
TCP	Transmission Control Protocol
TCP/IP	The Internet protocol suite
Test automation	Test which are run automatically at pre-determinate times
TFEL	Thin Film Electroluminescent
Unit test	Software testing method where individual units are tested
Valgrind	Dynamic code analysis software

XML	Extensible Markup Language
YAML	YAML Ain’t Markup Language

1. INTRODUCTION

The project is created for the use of Pilkington Automotive Oy. The basic idea is to create software and electronics to support the company's new idea to produce transparent displays inside a windshield. The project was initially introduced at Agri Technica trade fair in November 2015. The production is done in the collaboration of four factions Pilkington Automotive Oy is the client and manufacturer of the windshield, Beneq Oy produces the transparent displays and display control electronics, Tampere University of Technology (TUT) creates the software and additional electronics, and University of Tampere (UTA) uses the software and haptic feedback to create a demo into the trade fair.

The task of the Tampere University of Technology is to create electronics and software for the product which would support the upcoming exhibition with a demo. The demo combined the displays into target vehicle's CAN bus and University of Tampere's haptic feedback solutions. The target vehicle is chosen to be a tractor. The tractor's CAN bus messages are recorded and the demo is created on top of the real tractor CAN messages. The haptic feedback system contained vibrating seat, gesture navigation with a Leap Motion infrared motion sensor and steering wheels and pedals. The University of Tampere also manufactured lightweight Android application which is used to control the display over WLAN connection. The project overview is presented in Figure 1.

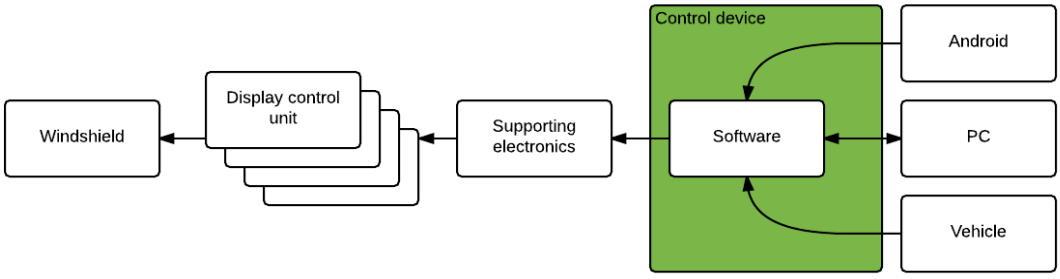


Figure 1: *The project overview*

Tampere University of Technology has three main tasks: the electronics which made the display usage possible, a software which supported the usage of the system and a PC control and test software to allow users to maintain the system. The main software can also be split into three independent sections: Linux driver module, graphical engine and

control unit. This thesis focuses on the graphical engine, Linux device driver and PC control software.

The main problem is the graphical engine's architecture that must support a different number of displays which have a different number of segments that are light in an easy to use interface for the client's needs. The graphical engine is targeting for the exhibition demo but the design must support the future applications and provide a basic structure that would provide a possible solution for the product. This thesis evaluates and tests the software with the exhibition in mind, so it does not address the product development.

The second problem affecting this study is the evaluation of the provided technologies (mainly the displays) and how they work in the windshield of a target vehicle. The evaluation is more about the physical aspects and the how the current electronics are working in a simulated real world environment. The last study problem is about how the new suggestions would change the current system architecture.

2. BACKGROUND AND THEORY

The main goal of the project is to create a software and supporting electronics for the Pilkington Automotive Oy's innovation to embed multiple electroluminescent displays inside a windshield. The project has a working prototype windshield which is embedded with four different types of a transparent displays. The transparent display uses SPI (Serial Peripheral Interface) communication protocol to communicate with the other hardware but because the characteristics of the SPI bus, it requires additional electronics to support the signal split into the four displays. The software is created into a single board Linux computer that can be controlled through a TCP/IP (Transmission Control Protocol/ Internet Protocol) connection. The Linux computer is also attached to the vehicles CAN bus (Controller Area Network) where it reads the vehicle's diagnostic data. The diagnostic data can affect the information displayed on the display unit.

The created software chain builds up from a Linux driver module, a display engine and a control unit. The projects software overview is presented in Figure 2. The Linux driver module handles the low-level SPI communication between the four display control boards and adds some higher-level features into the driver interface, which helps to ease some of the operations inside the display engine. The display engine contains all the graphical logic providing even higher level interface for the control unit. The graphical logic also handles the possible error situations, privilege problems and the timed operations. The engine allows the control unit to edit the displays with easy-to-understand commands that consider only the visual presentation.

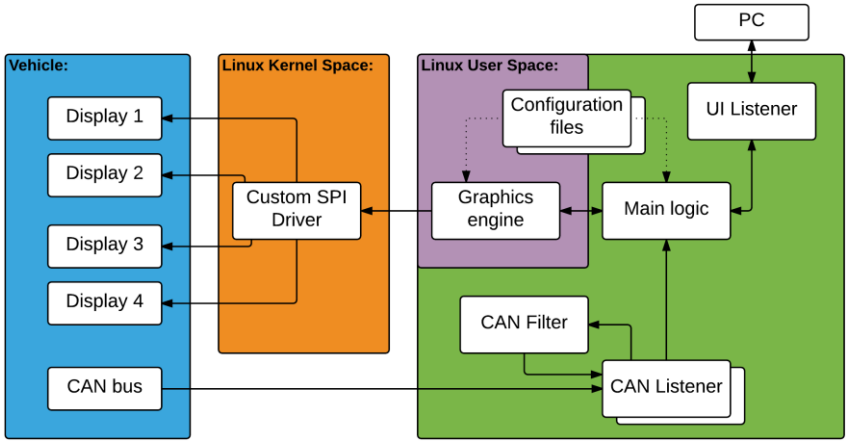


Figure 2: *The project's software overview*

The control unit controls the system main logic and the communication between the vehicle and the user input devices. The idea of splitting the graphical aspect into a separate section was proposed, because this way the whole system would be simpler. The control unit listens the vehicle's one CAN bus or multiple CAN buses and one or multiple TCP/IP connections for the user input devices. The control unit has a built-in server which listens certain ports and passes user commands to the graphical engine if they do not interfere with the CAN messages. The control unit also relays error messages from the CAN and the graphical unit to the user input device through the same TCP/IP connection. There is a separate master's thesis made concerning the control unit. [3]

A major part of the project is to develop an environment that changed during the development process radically. At the beginning, the team has two Linux based computers and an Apache Subversion (SVN) software versioning and revision control system. The project required a better cross translation environment so it is made on top of a virtual operating system. When the development started to merge, the development environment is combined with a router, which allowed all the team members to test the changes with their own computers. The development environment has an oscilloscope attached to it so it is easy to debug electronical faults. The structure of the development environment is presented in Figure 3. The development environment tools are a QT creator for rapid developing, a GNU Project debugger (GDB) for debugging, and a Secure Programming lint (Splint) for the static testing.

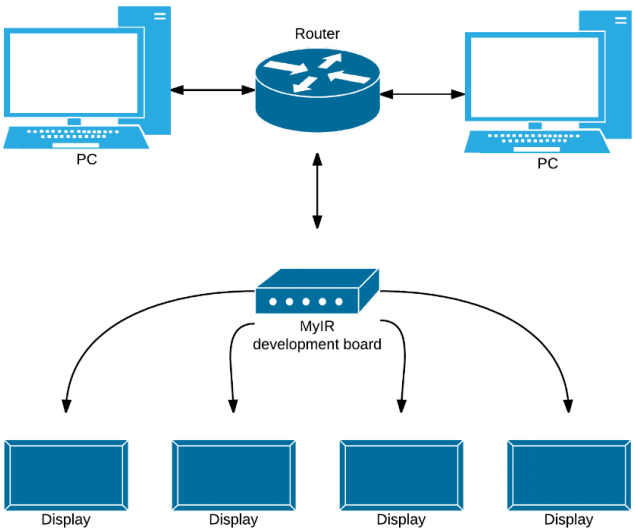


Figure 3: *The structure of the development environment*

2.1 The hardware

The hardware consists of various solutions. The main system is Linux development board with additional custom hardware made for the SPI communication between the displays and the development board (the custom hardware is excluded from this thesis). This chapter introduces the display's technology, Linux development board, and the SPI bus which is used for the communication between the development board and the electroluminescent displays.

2.1.1 Target system

The criteria for the Linux development board is that it contained at least one SPI bus, an Ethernet bus, and one CAN bus. The board must work in outdoor conditions, so the board should work in lower temperatures than they normally are designed for. There are multiple options for the desired development board, but the outdoor condition criteria reduced the number to only a few development boards.

The chosen development board was MYIR MYD-AM335X which is presented in Figure 4. The board contains one gigahertz Texas Instruments AM3352 processor and the default Linux kernel version 3.2.0. The device contained up to two SPI busses, one CAN bus, two Ethernet sockets, four USBs and a serial port for debugging. The board has 512MB of DDR3 SDRAM and 512MB Nand Flash storage. The board has a developer version so the control unit comes with a pre-made development shield that contains all the necessary adapters except for the SPI which is provided inside the pin rows. The board has optional harsh environment supporting, which made the board usable at the temperature range from minus 40 degrees to plus 85 degrees of Celsius. The board's kernel's default CAN library is edited to support CAN standard J1939 natively so the control unit software would be simpler.

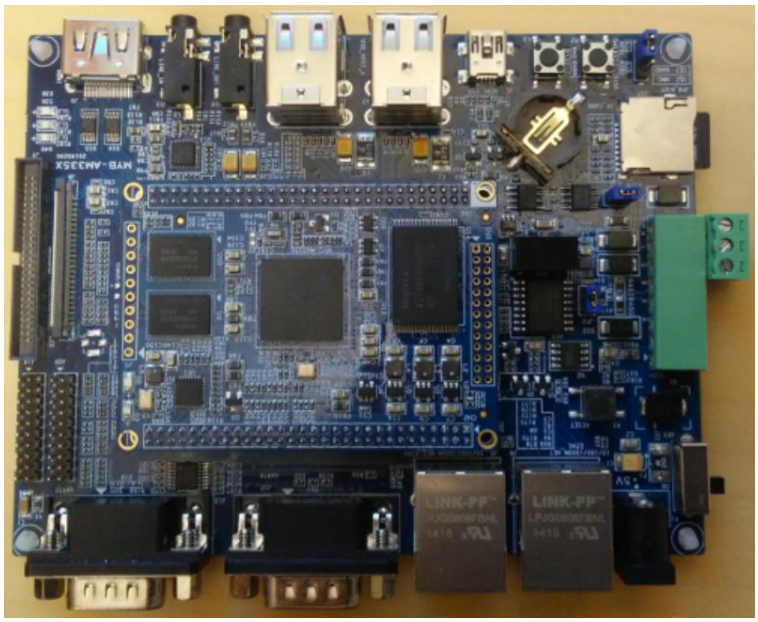


Figure 4: *The MyIR MYD-AM335X development board*

The main problem with this development board is the outdated kernel version. When developing the system, the newest kernel version is 4.3.0. The problem with this is that the SPI library's support for the usage of general-purpose input/output (GPIO) pins as chip selects (CS) is missing. Therefore, a driver needs to be created and the common SPI device (SPIDEV) driver module could not be used for this purpose.

2.1.2 Beneq Lumineq displays

The prototype contains four Beneq's Lumineq Thin Film Electroluminescent (TFEL) displays embedded inside a windshield. TFEL display is designed for extreme conditions, which makes them ideal for the demanding embedded applications. The displays are reliable and robust; they withstand temperatures from minus 60 degrees to plus 105 degrees of Celsius and have enhanced shock and vibration resistance. The displays have 179-degree horizontal and vertical viewing angles and keep over 85 percent luminance for the first 100 000 hours without any motion blur. [2]

The TFEL glass panel consists of a luminescent phosphorous layer in between of transparent dielectric layers, and a matrix of row and column electrodes. A single pixel on a display is lit by applying voltage to the electrodes, causing the area of intersection to emit light as show in Figure 5. The screen is controlled with a circuit board that is connected to the screens with a flexible flat cable. The circuit board uses a SPI bus to communicate with other devices. [7]

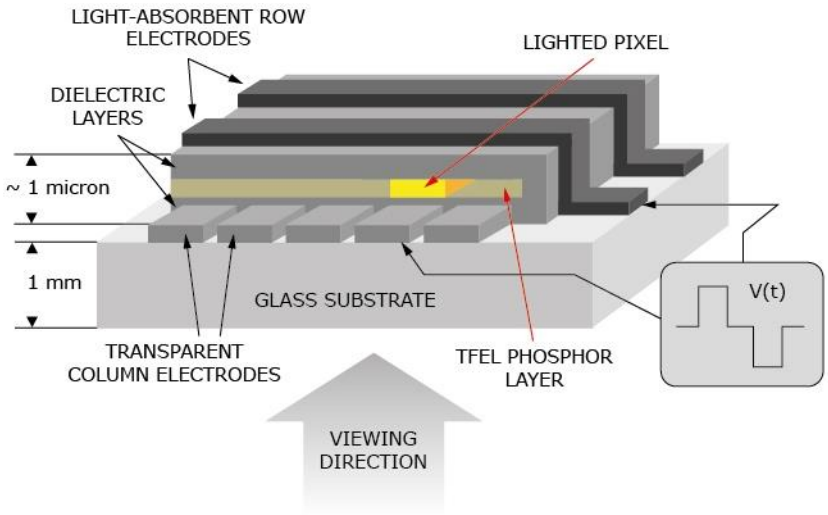


Figure 5: *How the TFEL display operates [7]*

The display's control electronics provide basic functions for the screen control. The board has a self-test pin reserved for the user to test that the screen is working properly. The SPI messages provide a set of commands that are used to set a single segment on or off. Additionally, the screen's luminosity can be changed by dimming the screen by changing the voltage level or the frame rate count. The Lumineq control board's frame rate range is from zero to 2047, but at the testing phase of this prototype, project showed that if the frame rate decreases to below 500, the human eye can see the screen blink.

The voltage dimming is featured within this prototype, but its use is not recommended because it does not work properly. When lowering the voltage value, the screen's segments become unequally visible. The segments in the middle of the screen have normal brightness but the ones at the edges have a gradient luminosity that does not look good in the eyes of the user.

The testing environment has four different screens which could contain up to 40 segments each (in this project they are referred as lamps), but mostly there are around 20 segments in one display. The segment division is presented in Appendix A. Basically the segments are one part of a screen that can be set on or off. They can also form an icon or a group. For instance, the Bluetooth icon contains two segments: the outer ring and the Bluetooth logo.

The segments are used to form 7-segment displays, icons, arrows, or parts of a vehicle. The displays are divided into four different groups: communication display, vehicle display, distance display, and overall information display. The displays are presented in Figures 6 to 9. The display layout is selected by Pilkington Automotive Oy for the exhi-

bitions where they wish to present a large set of possibilities of where the product can be used.

Figure 6: *The first display: Communication*

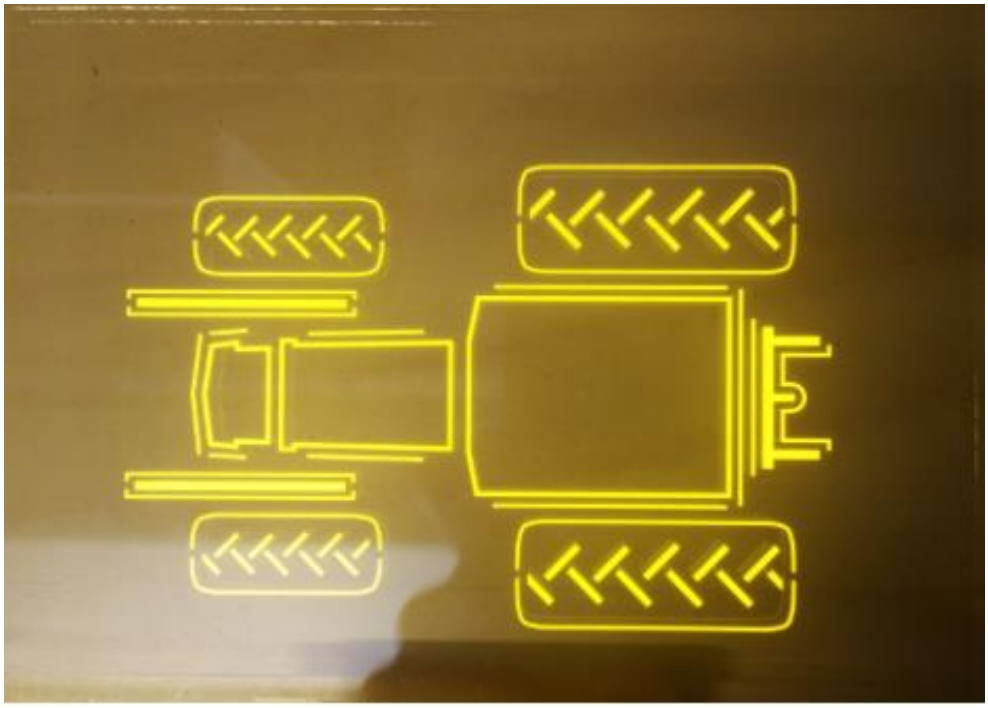


Figure 7: *The second display: Diagnostic*

Figure 8: *The third display: Distance*

Figure 9: *The fourth display: Vehicle state*

2.1.3 SPI bus

Serial Peripheral Interface (SPI) is a bus used to send and receive data between microcontrollers and small peripherals. Commonly the applications are Secure Digital cards (SD) or liquid crystal displays (LCD), just to name a few. SPI bus is restricted to be a single master bus, but it can have multiple slaves depending on the number of available slave select (SS) lines. The SPI is not a robust data transfer bus so the data transfer can become easily corrupted. A way to prevent corruption is to keep cables as short as possible. [4]

SPI bus is synchronous, which means that it has separate lines for data and clock. The clock line (SCLK) keeps both ends synchronized. The desired slave selects (SS) are chosen with a unique SS line, and the data is transferred from master to slave with a master-out-slave-in line (MOSI). There is also a possibility that a slave sends data back to the master with a master-in-slave-out line (MISO), which is not used within this project. Usually the SPI bus does have only two chip selects, but the chip select system can be expanded with GPIO pins if needed. Figure 10 represents a typical line connection of the SPI bus. Master is able to configure the clock polarity (CPOL) and clock phase (CPHA). CPHA controls the data's read position i.e., is the data read on the rising or on the falling edge. Data transfer process is presented in Figure 11. [6]

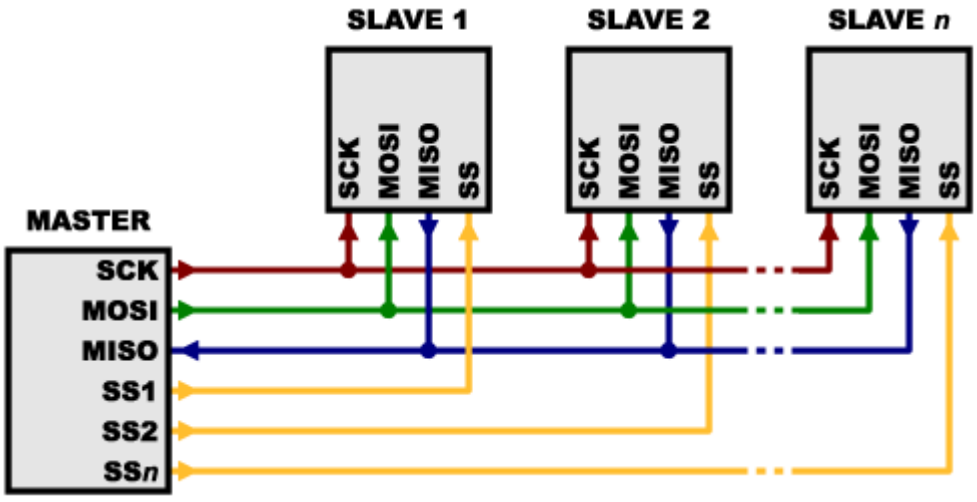


Figure 10: *Typical SPI bus usage [4]*

CPOL	CPHA	Read on …	Off status is …
FALSE	FALSE	Rising edge	LOW
	TRUE	Falling edge	LOW
TRUE	FALSE	Falling edge	HIGH
	TRUE	Rising edge	HIGH

Figure 11: *SPI data transfer timing diagram*

The slave select lines are set to HIGH state when the lines are free from data transfer. When the transfer to the selected device is started, the value of the device's slave select is set to LOW. When the transfer is finished, the line resets to HIGH. The SPI master has additional options for the data transfer. The most common configuration is the frequency of the data transfer. Usually the transfer's maximum value is a few megahertz, but when the value is lowered, the data transfer becomes more reliable and robust. The word size can vary depending on how the electronic device is configured: typical word sizes are an 8-bit, a 12-bit, and a 16-bit. [6]

2.2 The software

The software is mostly custom. The system uses only standard libraries and one third party library. The software is built into two different processor modes: the kernel and the user space. This section introduces the third party YAML library, and a small portion of the Linux kernel device driver module.

2.2.1 YAML markup language

YAML Ain't Markup Language (YAML) is a data serialisation language designed by Clark Evans, Ingy döt Net and Oren Ben-Kiki in 2001. The ideology behind YAML is to create both human readable and computationally powerful language to simplify extensible Markup Language (XML) and bring YAML into compliance with JavaScript Object Notation (JSON). YAML has reached its 3rd edition (YAML version 1.2) and it is used in configuration files, Internet messaging, data auditing and in other similar uses. Example of a configuration file is presented in Appendix B. [8][9]

This project employs YAML as a configuration language. YAML was chosen to the project because it is readable and the file sizes are smaller than of any of its competitors. The support for different languages is good. The project uses LibYAML parser library which is made with C programming language and which reads 2nd generation YAML files. LibYAML is an open source YAML parsing library written by Kirill Simonov and it is developed for Python Software Foundation. LibYAML is widely used open source YAML parsing library within the Python community. [5]

2.2.2 Linux device driver module

The Linux device driver module is a set of code that is written into the Linux Kernel as a removable module, which allows Linux processes to utilise certain pieces of hardware. The main purpose of the driver module is to allow programmers to create a software layer for their needs so that the two driver modules for the same hardware can offer different capabilities or allow multiple processes use the same driver module to connect into a piece of hardware without any conflicts. The driver module also hides the hardware's functionality. The driver is part of the Linux kernel code, and therefore the driver module has a stricter limitation to its operating methods. The module might crash the whole system if it is not working properly. The driver has a mandatory license information. The drive module uses a GPL (General Public License) because the project uses Linux kernel's SPI driver interface which is under a GPL license. The GPL license must be inherited from the source. [1]

The module uses plug and play functionality with the SPI line which automatically activates and deactivates itself when the correct SPI device is attached into or detached from the development board's SPI line. This can be achieved by editing the kernel's processor specific file and adding a new SPI board info structure onto a list of board information that must contain the needed information so that Linux can automatically recognise the correct SPI device. The most important information consists of MODALIAS (name of the driver), the maximum speed, bus number, the amount of chip selects, and SPI mode.

The main purpose of the driver module is to provide basic manufacturer-specific functionality that allows the display engine to control the Lumineq displays. There are several extra functionalities added into the system, which allows more low-level operations within the drive module, such as negation of the segments.

The prototype must control four different displays. All the displays must be connected into unique chip selects, but the board has only two chip selects per SPI line so a custom chip select system is created over development board's GPIO pins as suggested in Figure 9. The problem with the kernel version 3.2.0 is that the kernel's SPI interface does

not support GPIO functionalities, so the system is done by changing the kernel’s registers.

3. FEATURES

This chapter describes the basic features of the system. The chapter starts with the introduction of the display engine, continuing to the driver, and ends with the configurations files. These features are the top level of the software. The sections contain only a global view and high level explanations of the features. This chapter is also presenting the high-level topologies on how the features are linked together.

3.1 Display engine architecture

The display engine can be divided into two separate parts, the configuration file reader and graphical logic. The display engine reads the configuration files at the beginning and then it continues to run the logic. The graphical logic manages three different tasks: it processes incoming commands, handles timed commands which have an execution time, or sleeps if there are no incoming commands or the timed commands which are executed now or later in the future.

The software starts automatically when the device is powered and loops until the device is closed. The system starts by reading the YAML configuration files to the memory and the software stops if the parsing fails. When the parsing is successful, the software starts the infinite loop which runs until the device is powered off. The engine's architecture is presented in Figure 12.

The main loop's first task is to read the next command of the incoming special FIFO (first-in, first-out) file's next command. If new command exists, the command is processed. After the command processing is finished, the loop continues to check if there are timed commands in the dynamic timed commands list, and compares the execution time with the current time. The timed command lists are processed until the list is empty or the next command has a timestamp that is greater than the current time. When the list is empty, or the timestamp is in the future, and there are no incoming commands at the FIFO list, the program sleeps for ten milliseconds before trying again. The program does not reserve unnecessary resources from the system if they are not needed.

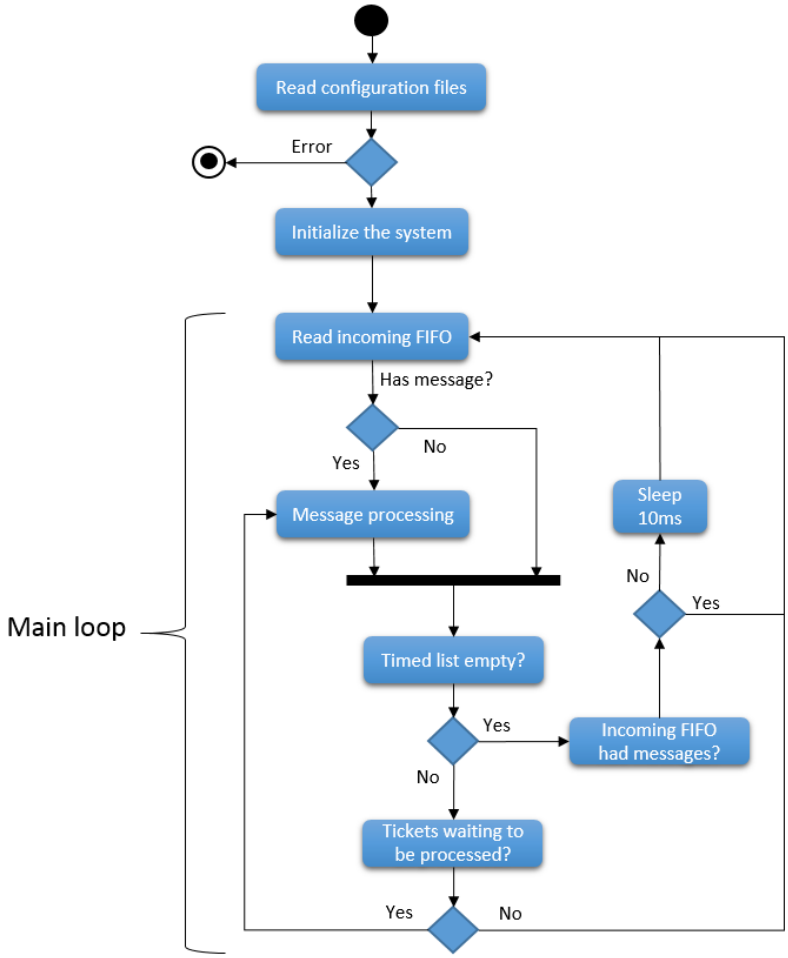


Figure 12: *The engine flow chart*

The system transforms all the incoming commands into tickets. A ticket is a structure which holds information about the command type, target ID number, whether it needs a return message, and additional data. The command types are specified in Figure 13, the ID is self-explanatory, and the return value is needed when the main logic needs acknowledgement from successful operation. The additional information depends on the command type, but there are only two alternatives: The digit commands have a numeric value to be displayed, for instance “12:50”, and the rest of the commands use special indication which sets the commands type to off state (0), on state (255) or blinking (1-254). The blinking value tells how long an item is on and off, and the value is multiplied by ten milliseconds. This information is used inside the graphical logic to do all the necessary operations.

Command	Definition
0: Unused command	Reserved for internal use
1: Screen command	Set screen on / off / blink
2: Icon command	Set icon on / off / blink
3: Lamp command	Set segment on / off / blink
4: Group command	Set group on / off / blink
5: Digit command	Set digit group into desired value
6: Framerate dimming command	Set framerate dimming into 0 - 2047
7: Voltage dimming command	Set voltage dimming into 0 – 255
8: Navigation command	Start or end navigation, or navigate next or previous screen or icon
9: Return command	Return acknowledgement or no acknowledgement back to the control unit
10: Debug command	Reserved for the user commands which provide a quick solution to test single units

Figure 13: *The list of commands*

There are also two additional fields within the ticket structure: a pointer to the next ticket and a timestamp value. The tickets are stored in dynamic list and therefore the tickets must contain a memory pointer to the next ticket item. The time value (or a timestamp) is used when the ticket is timed. Basically, this happens only when a blinking value is set. The system calculates a timestamp with a time in the future, and the tickets are processed when the ticket's timestamp has been expired.

3.2 Display engine features

The YAML parsing functions are separated depending on the file which is parsed, each file has its own function which handles the parsing. This solution makes the code more logical and safe, changing a single file structure and its parsing system, will not break

the other functions. The parser also has a debug mode. The mode prints the output into the console, which helps the client to see if the configuration files has any errors.

The command processing system can be divided into common processing, digit value processing, and into two special operations, the navigation mode and timed ticket processing (blinking). The common command processing is done with the following steps: First a loop runs and checks all the segments which are needed in the common command (for example the icon can be formed from multiple segments) and tests that the new command has equal or larger privilege level than the current one. At this point, if any of the segments contain larger privilege level, then an error is returned and command ticket destroyed.

When privilege check passes the system starts to clean the old commands out of the system, before pushing a new one. This must be done because the new command might contain a set of segments which are overlapping multiple commands. Therefore, there might be a larger number of segments which are removed before new is pushed into the system. Cleaning is done by reading the segment status values from each segment individually. Each of the segment status values combined with the type tell the system which ID and type of an item it has before. This data is loaded from the list of items and all the item’s segments are released. When the old item has an additional timed ticket in the timed ticket list, it is also removed from the list.

After the cleaning operation is done, the new updates are pushed into the internal lists and the updates are pushed into the displays. If the ticket is blinking, then the ticket is added into the timed ticket list. This process is detailed in Figure 14.

Timed ticket variants are processed after the incoming messages are processed. When the timestamp of the first ticket on the timed ticket list is lower than the current time, the ticket is processed as a normal ticket. Timed tickets are created if the incoming command contains a blinking value, then the system processes the tickets as normal tickets, but adds the same ticket back to the timed list with updated timestamp. The system allows also the system to add other than blink tickets with timestamp, and after they are processed, the tickets are destroyed.

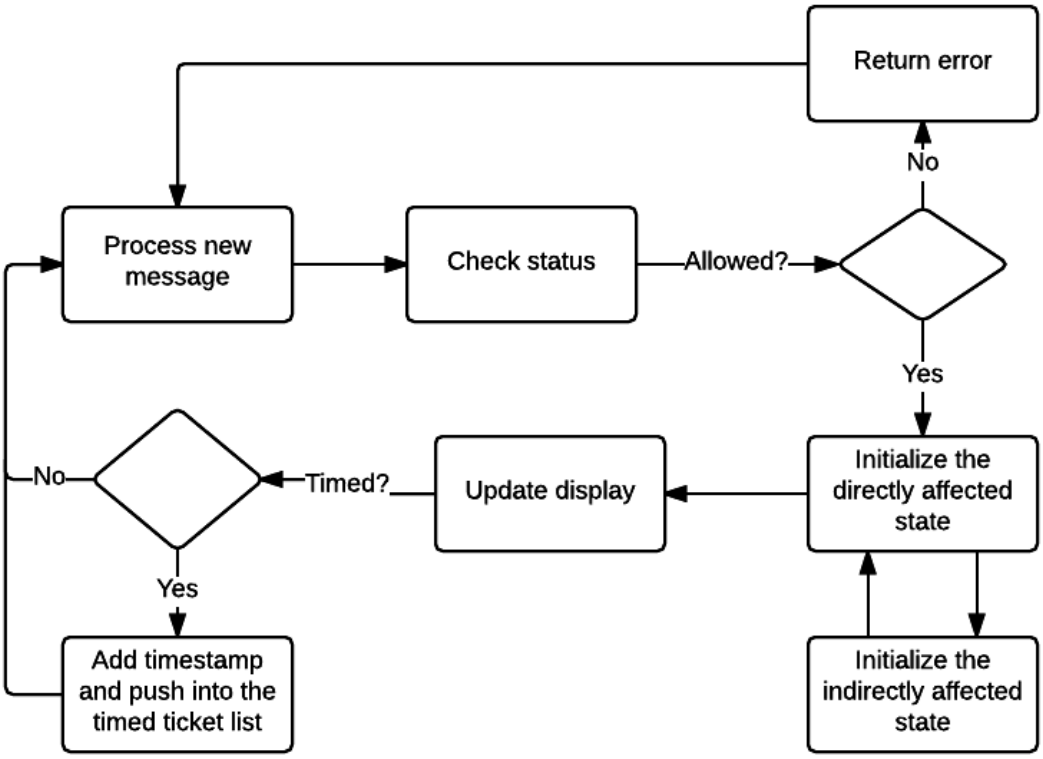


Figure 14: *The normal ticket processing*

The engine has a feature which allows the user to navigate and select icons which affect the on-screen information. For instance, when the information display shows oil pressure in the display's digit group, the user can change it to rotation speed with the navigation. This is a tool for the control unit to have a way for the user to make selections and to get visual feedback. The navigation system is handled by gestures done over the Leap Motion infrared sensor. The system has five gestures. A hand swipe from left to right or vice versa will activate the screen navigation mode. The gesture will change the navigation status into a screen selection and starts the adjacent screen of the currently selected screen to blink (depending on the direction of the gesture). The next two gestures happen when user circulates a single finger in a small circle over the Leap Motion. This makes the icon selection which starts to blink the currently selected icon's adjacent icon (again depending on the direction of the gesture). The last gesture is the selecting gesture which is done by pushing the air with a single finger. In the icon selection, this makes the selected icon as the currently active one and with the screen selection, the first icon of the screen will be selected as the active one.

The navigation system is automated depending on the icon information that is passed with the icon configuration file. The icons are designed because there are multiple numeric values which can be passed for the user but currently the displays have only two numeric digit values that can be presented simultaneously. Therefore, the user can select

items on screen which affects the data displayed in the seven segment displays. The navigation mode clears screens and starts to blink the active icon. The active icon state can be changed with five commands which form the navigation control. Previous or next changes the current actively selected icon. “Previous screen” or “next screen” commands change the active icon to be the first icon of the next or previous screen. The last command is “Select” which resets the displays to the state where they were before navigation. When the navigation is on, the incoming tickets are stored into temporary dynamic list which is processed when the navigation is set off. This way the updates are running at the background, and the user receives the latest updates when the navigation ends.

Digit tickets show numeric values to the user. They are prioritised over other command types when the navigation mode is off. This privilege is set because the information from the vehicle must be surpassed to the user with highest priority: for example, the vehicle’s distance from an object which needs to be constantly and accurately shown to the user to prevent the user from crashing into the object. Another reason is that the digit number creation process manages the largest set of commands on the displays and therefore the digit values skip the ticket validation process and always are pushed into the display. The digit process always run the clean-up sequence to ensure that there are no overlaps between other commands. When the navigation mode is on, the tickets are deleted immediately, they would fill the memory. Because of this, the old tickets become outdated almost instantly. The problem with the privilege digit access is that when the digits are needed for other purposes, the system must end the numeric digit feedback from the control unit. Another problem comes with the navigation. If some values must be preserved for a longer period, they will be lost as the system does not currently store the digit data. A solution for this problem is a new command type which would work almost like the digit command but which would not be destroyed in the navigation mode.

3.3 Driver architecture and features

The Lumineq control board has three main features which allows are used to change the display segment’s state. The main feature is to set a segment on or off. The segments are divided into four segment groups, where the first three groups are editing twelve segments at a time and the last group controls only the remaining four segments. There are also two different ways to change the dimming value of a single display through editing the frame rate or voltage.

The driver module has additional set of features which add a simple set of functionalities which would be more difficult or inefficient to make at the graphics engine. The timing is the largest concern when updating the screens. There are three different com-

mands which help the system to time the updates: push, edit and update. When user wants to change a state of a single segment, it usually must be updated straight away, and the push command changes the screen items in the internal driver state list and pushes the changes instantly to the display. When updating multiple segments, it is better that the engine does all the necessary changes before the screens are updated, so the edit command only changes the segment state within the driver, and the update command will push the edited state into the displays. The three-way system speeds up the updating process and updates the visuals at once instead of slowly changing the state one segment at a time.

There are custom features which affect a single display. These functions allow the engine to set all the display's segments on, off, or blinking at the same time, so the engine can just pass a concise set of information quickly with one command. In addition to that, there is a command which clears all the displays from the system. Basically, resets the whole driver memory quickly. This option is used to close the system. The system still works in the background but the displays are transparent for the user. The driver's architecture is presented in Figure 15.

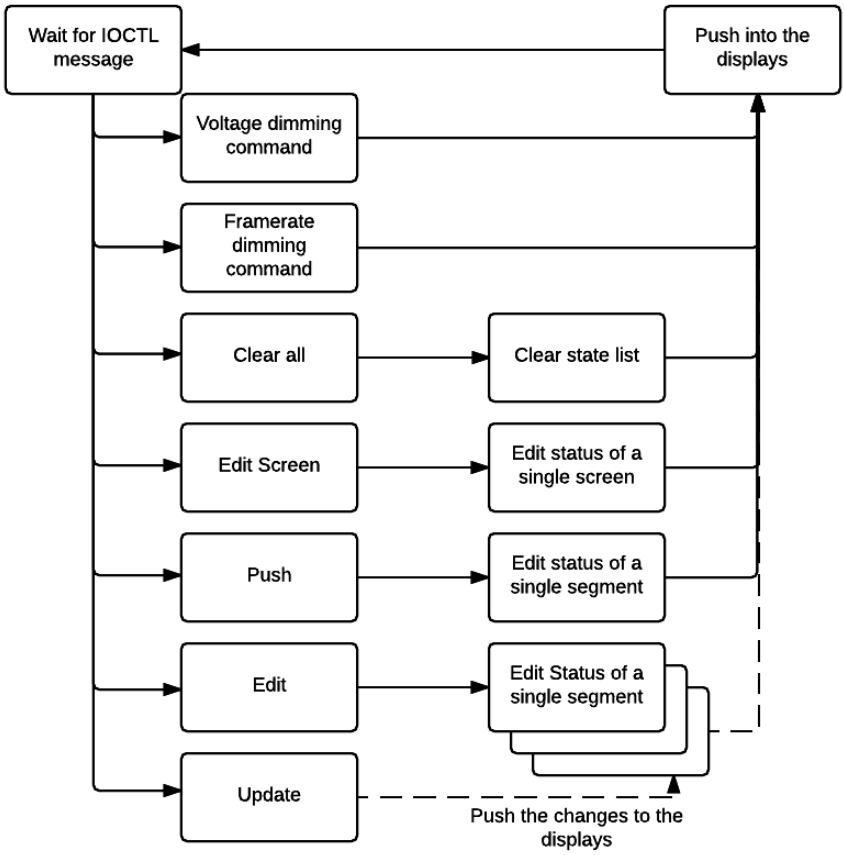


Figure 15: *The driver architecture*

3.4 Configuration files and parser

The configuration files are split into separate files depending on the content. There are total of five files: setup, lamps, icons, groups, and digits. The two main files hold the information about the global setup and the information about the segments (referred as lamps in the project). The data from these files are affected by the Lumineq displays. The last three files (groups, icons, and digits) form the corresponding items from the segment (lamp) information and they are purely a representation of an engine feature. The files are separate, because it makes the parser code and the user input simple and readable, and eases up the update process for the user. The data is loaded into a dynamic list of custom structures which are passed into the main loop.

The global setup contains value ranges (dimming values) and a set of global values which affect the main loop process (blink and level values). Segment file holds the information about the target display number and the Beneq's pre-defined segment number of that display (listed on appendix A). Groups and icons hold the list of segments which form the group or the icon. Digit groups hold the information about the individual digits and the digits hold the information of the segments which form a single digit. The digit types are a 7-sequence digit (7 segments), dot or double dot (single segment) and zero (single segment). The accurate data is presented in Figure 16.

File	Field	Content	Definition
Lamps	lname	String	Unique name of a lamp
	scrid	Numeric	Screen ID where the lamp is located
	lmpid	Numeric	Physical lamp number
Icons	iname	String	Unique name of an icon
	ilamps	List	List of lamps in icon
Groups	gname	String	Unique name of a group
	glamps	List	List of lamps in group
Digit group	dgname	String	Unique name of a digit group
	digits	List	List of digits in a digit group
Digit	dtype	String	Type of digit (7, single or zero)
	gdigits	String	String of group which holds the digit's segments
Setup	maxvoltage	Numeric	The maximum voltage value in the voltage dimming
	maxframerate	Numeric	The maximum framerate value in framerate dimming
	selectblink	Numeric	Length of the navigation blink ON/OFF (1 = 10ms)
	lamplevel	Numeric	Priority level of a lamp
	grouplevel	Numeric	Priority level of a group
	iconlevel	Numeric	Priority level of an icon
	screenlevel	Numeric	Priority level of an animation

Figure 16: *The YAML configuration data fields*

4. DRIVER MODULE

This chapter presents how the driver works. The chapter explains the major influences on why certain choices are made and why some choices are excluded from the initial design. The main factor is to explain how the driver works and how the custom features are done.

4.1 Initialization and uninitialization

The driver module works as a plug and play driver, which means that when the SPI line is attached to the Linux device's correct SPI line (with this project SPI 0), the kernel automatically detects the driver module and checks if the characteristics of the attached device are matching with the physical setup of the previously set SPI board information. The Linux kernel recognises the driver after it is initialized with the terminal command INSMOD (install module). At the beginning of the initialization, an initialisation function registers SPI driver device and character driver device into the Linux kernel. The SPI driver device handles the information transfer between the display's control device and the Linux driver module. The character device handles the communication between the driver module and the engine through IOCTL (input/output control) calls.

The initialization continues by changing the desired pins from the Linux hardware to work as a GPIO pins. This needs to be done because usually the pins have multiple purposes and they are not in GPIO mode by default. The old settings are saved before any initialization so the system can be restored into its original state after the SPI line is disconnected or the module is removed with RMMOD (remove module). The change to GPIO pins happens through the kernel's setup registers from where the pins are also set into output state.

The driver module needs to initialize the SPI line, and to do that the hardware needs to be recognised automatically. This is possible by changing SPI board information table inside the Linux kernel's source code. This information must be asserted into the kernel's processor specific C file before building the kernel. The main fields of the board information table are MODALIAS (unique name) and bus number. The bus number tells the driver about the SPI line pins where the custom SPI hardware will be attached. When the hardware is attached, the kernel passes the information automatically into the correct driver module which contains a same MODALIAS as the SPI board information table.

There are additional fields inside the SPI board information table: the number of chip selects used, maximum speed in hertz, and mode which tells the system which CPHA and CPOL setups are used. This project does not use the default chip selects, and as the CPOL and CPHA values are 0, the mode is zero. The maximum speed is set to be as low as possible because SPI has problems with longer cables. The Beneq's display controls can be as fast as three megahertz but the prototype has long cables (0.7 – 3 meters) so the speed of the line is radically decreased into ten kilohertz, which made the system more reliable with less errors in the transfer procedure.

The removal of the driver module is done by detaching the hardware or by unregistering it with RMMOD. The uninitialization process starts by closing the screens with "clear all" command. Then the driver module starts to do its own uninitialization process. The first process layer of unitialization happens in both cases. The allocated memory is freed and the SPI driver device is unregistered. The second layer happens only when the RMMOD command is used. The system begins with the first layer, but after that the device frees all used data, unregisters the character driver device, and sets the system to its original state which is stored at the initialization.

4.2 Communication

The communication between the engine and the driver module is created with an IOCTL call. The kernel uses the IOCTL as a gateway from the user space to kernel space and it works like FIFO files. IOCTL uses a shared file to pass the information between the two spaces: when data is pushed into the IOCTL file, the kernel will inform the driver module about the new content. This project uses only one way IOCTL because the driver module does not need to pass any information back into the display engine. The IOCTL call is common, and it is used with many of the driver modules inside the Linux kernel. A certain call needs to have a unique major number and minor number. The idea is to choose a set of two numbers which both are eight bits wide and where the combination of the two values are different from the numbers which the other driver modules use so the modules would not overlap the messages [1]. Chosen major number for this project is 120 and for the minor number it is 88 which are both unreserved by the target hardware. The data is passed with a custom struct message which contains four numeric values: display ID, segment ID, on/off/reversed value, and a unique driver status (for example, update or edit).

The kernel's SPI library does not support GPIO pins as chip selects. Selecting the correct display needs to be done manually by changing the kernel's setup register before and after the commands are passed into the displays. Before the data transfer, the system changes the specified GPIO pin OFF which tells to the specified control board that a message will soon be transferred. After a short delay, the data transfer will commence,

which is followed by a second short delay to inform the display's control board that the transfer is completed. The system sets the GPIO pin back to ON state which means that the SPI line is free from any transfers. The timing of the data transfer is presented at Appendix C on page 6. The register data fields are stored depending on the chosen GPIO pins, and the system chooses the correct pin per the display's ID number. If the GPIO pins need to be changed, or there are more than four displays, the driver code needs to be updated and rebuilt. This manual change would not be necessary if the kernel's SPI library has native support for the GPIO pins as chip selects.

SPI communication between the driver and the displays is designed by Beneq. One message is 12 bits long, while the highest four bits are the commands header and the remaining bits are the data. The list of how the commands are built is presented at Appendix C on page 10. The control of the segments requires two values: the header and a list of segments. The header is the number of the list of segments (from zero to three). A list of segments is formed from each segment as a bit value in a data field resembling as an ON or an OFF state. When updating the segments, the driver must remember the statuses of all the segments of every display.

The Lumineq control has also two ways to dim the displays, and they have unique header commands (see Appendix C page 10). The dimming values are from zero to maximum. The maximum values are 255 in voltage dimming and 2047 in framerate dimming. The driver module supports the voltage dimming although it is not advised to use it within this project, and its usage is not recommended by the Beneq, either. The voltage dimming makes the segments to light unevenly.

4.3 Commanding the screen

A single screen has a maximum of 40 segments and all the segments are grouped into four lists of segments. First three lists have 12 segments, and the last one has the remaining four. The driver module holds an array of bytes which is needed to track the state of a single segment. The size of the array depends on how many displays there are in the system. This prototype's segment list size is four displays multiplied by 40 segments. The array holds the on-screen information of each segment in the system. The segments' array value is zero, then the segment is OFF, and if the value is one, then the segment is ON.

The graphics engine sends the segment updates with the same status that it uses within its internal processes. The zero value means off state and 255 means on state. The graphics engine blinks a segment for a certain period when passing values from one to 254. The driver module holds the array of the on-screen state, and when the blinking value is passed into the driver module from the graphics engine, the segment is negated.

This way it is not necessary to make additional changes into the state value before the engine sends the ticket. The negation makes the overall procedure simpler, because the graphics engine does not need to deduct, or keep a state list of the segments, or ask the state from the driver.

The module allows the engine to have additional functions on controlling a whole screen with one command. The screen update has the same three levels as the single segment control (on, off or negated). The difference is in the display blinking. The solution to blink a display is to read the value of the first segment from the displays array of segments and to negate that value. This value is passed to every segment of the same screen. Usually the segments are given an unique ID, but the module has a custom ID for the screens ranging from one to four. The custom ID zero is reserved for the future if there is a need to control all the displays simultaneously. There is also a command which clears system's all displays. This command works as a system shutdown for the user. The command uses the same command logic as the screen command but it also resets the segment array.

The module has an additional array for holding information about the segment lists where the segments are referred as single bytes. The array tells the module which segment lists are changed after the last update that has been sent to the displays. This feature allows the engine to make multiple changes (edit) to the system before updating the displays with an IOCTL update call. When the output is pushed into the displays, the user will see only one update on the displays. The size of the array is the number of screens multiplied with number of segment lists which is four. When the incoming IOCTL command type from the engine is "edit", the changes are done into the array of segments, but the driver module also changes the array of lists. When the next IOCTL update command is passed, the driver forms SPI commands from all the lists that are edited after the latest update.

5. ENGINE

This chapter presents how the display engine works. The chapter explains the major reasons on why certain choices are made and why some choices are excluded from the initial design. The main purpose is to explain how the display engine communicates, how tests manage updates and tickets, and how errors are handled. This chapter also focuses on some major features that are at the centre of the system, for instance timing and navigation.

5.1 Communication with control unit

The two-way communication between the engine and the control unit are built on top of a FIFO file and the communication between the engine and the driver is built over an IOCTL. Both communication methods have different messaging systems which are supporting the whole graphical update process. The FIFO messages are designed to be higher level than the IOCTL messages which supports the system's main purpose that is to send the minimum number of messages through the system into the displays.

The commands presented in Figure 13 on page 16 are the different messages that are used in communication between the control unit and the display engine. The main logic needs to pass a command and target ID, and data which in most cases is a status value (0-255) or a digit (this data is specified in Appendix D section 1.1). There are a few special cases where the pattern breaks. The navigation mode combines the type of command and the on/off value: the first bit is the on/off value, and the last seven bits are the type ID of the navigation mode. The last seven bits tells the system about which navigation mode is needed: the screen navigation or the icon navigation.

The second special case is the digit command. The data values are expanded into the total of eight bytes where one byte represents a single digit. The system is created to support digit values which are in the form of "xx:xx:xx." The reader compares a single character with a comparison list that tells which segments of the seven segment displays are needed to light on the screen when a certain character is presented in a seven-segment display. This means that the single digits need to be constructed segment by segment from top to bottom and left to right.

The last two special cases are the debug and return commands. The debug command pushes the segment value straight into a display by ignoring all the tests and status

changes which normally it would go through. The difference is that the ID value is split into two values, the HIGH byte is the display number and the LOW byte is the segment ID on the chosen display. The last special case is the return value. These values are used with two cases: when the command requires acknowledged (ACK), or when an error occurs. The messages ID field is the return type of the error or ACK message. The rest of the command depends on the type of the message. The types and contents are presented in Appendix D.

5.2 Ticket processing and communication with the driver

The ticket processing can be split into three cases. The two special cases are navigation and digit messages, which are explained later, but all the other tickets are processed with a single test and initialization process. The ticket's privilege is always tested and an initialization is executed when necessary. After a successful test, the send process changes the old status state to a new state and breaks the ticket into one or multiple IOCTL packets which are pushed to the driver.

The driver special cases are pushed straight into the system. These special cases are the dimming and debug commands. The screen is also pushed straight into the displays, but this command needs also to edit the sequence status list. These special commands have a unique send function which transforms a ticket into an IOCTL message.

The segment, group and icon commands are pushed into the driver through a sequence of functions that affect the display engine. The process starts by the update of the segment status list which loops through all the segment items on the list and changes their statuses. After a single status change, the segments are updated by segment IOCTL function which transforms a single segment information into the driver message form. The single segment update uses the driver edit and update command, but with an icon or group, the segments are sent with driver edit command and when the last segment command is sent, the driver update command is sent. The process is presented in Figure 17. The problem with this system is that it does not contain a feature which could reverse the update if something goes wrong after some of the commands have been pushed into the driver.

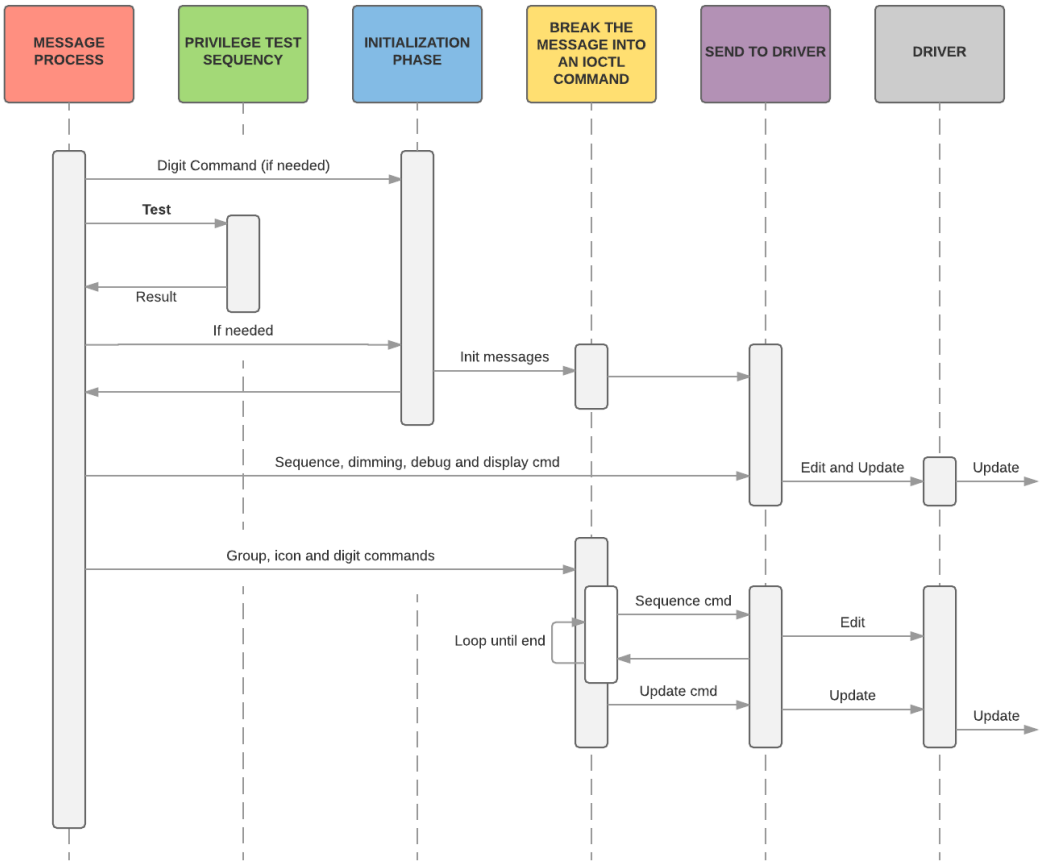


Figure 17: *The message handling process*

5.2.1 Command privilege checking process and initialization

All the display update commands (except the special cases) start with a privilege check process. The process compares the new changes into the up-to-date segment status list and returns six different return status values which tell the logic how to proceed. There are three error values that block the processing of the new command and three status values which the logic uses to perform certain actions.

The first error is the “no priority” error which is affected by the level values from the setup configuration file. When the new command’s type has a lower level value than the old one, the progression is stopped, and a corresponding not acknowledged (NACK) is returned. The second error occurs when the new command’s state change is the same as the old state. The “no change” error is not actually an error, but if the state will not change, there is no need to push the new change into the displays. The third error occurs when there is an error within the test process.

The first status value is that the new command is overlapping itself. This means that no further actions are needed, and the process can continue to the send process. The second status value is that the item is overlapping with multiple items. This makes the logic to go into initialization phase where all the overlapping items are nulled on the status list and the display before the new command is sent to the send process. The third status value tells the logic that the value contains timed items. The logic removes all the timed items from the timed list, before it starts the same initialization phase as with the second status value.

The segments are compared with a single item on the status list but the screens, icons and groups are tested by looping all the items which are affected by the new status changes. The check sequence starts by comparing the old status priority with a new one, and if it is higher, it continues by returning the “no access” value. The sequence proceeds with comparing the old command type and data with new ones: if they are the same, the ”no change” value is returned. Next, the sequence tests the old type with the command, and if they do not match, the overlapping value is returned, which triggers the initialization. The last test concerns the data. When the data is between zero and 255, the system contains timed tickets, and the check process returns timed item values. The test and initialization process is explained in Figure 18.

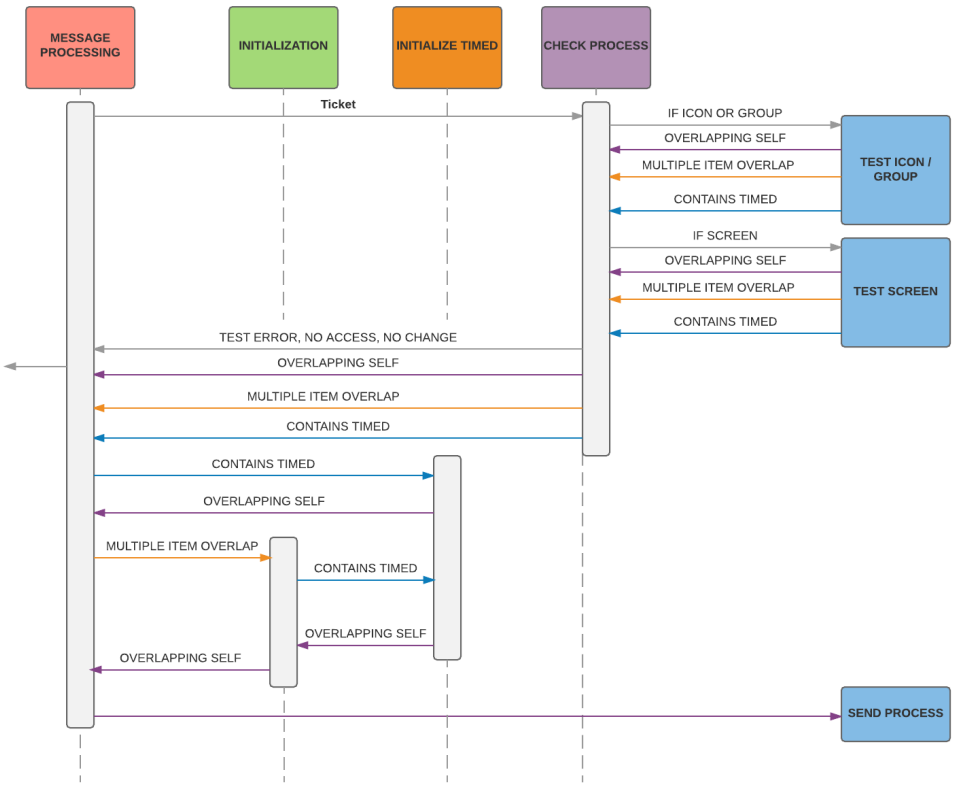


Figure 18: *The privilege checking and initialization process*

The initialization process loops through all the segments which are affected by the new command. The process reads the old state and depending on the states, nulls all the values which the old item has changed. When the old item is a segment, the system nulls the segment. When the old item is a group, an icon or a display, the system nulls all the segments which are connected to the item. The timed items are also removed from the timed list.

5.2.2 Timing

The timing system is created to allow the user to set segments to blink, or to process a command after a certain amount of time has passed. This feature is particularly important because the prototype can display only yellow light. In some user cases the system needs to display important messages like errors or faults inside the vehicle. In addition, the timing system allows the user to get feedback of the user control while navigating. The timing system allows all kinds of timed operations, and it can be used with every command, for instance for creating an animation system later.

The timing works with a single dynamic list of tickets. The tickets hold the timestamp of the execution time, that are used in keeping the timed list in order. The timing list is tested after the loop has processed the next available FIFO command if there are any. The process time of the first ticket on the timing list is compared to the current time, and if it is lower or equal, the ticket is processed and the timing list is reorganised. This comparison loops until the list's first item's process time is greater than the current time. When the time loop ends, the main loop starts from the beginning. The main loop process is presented in Figure 12.

The time data is stored into a struct which is defined at the operating system's time library. The struct has two long values: one for seconds and one for milliseconds. This way the timed functions can be set as low as one millisecond, but the projects minimum time is decided to be ten milliseconds. With this prototype, a single byte is set to represent the segment's status. When it is zero, the segment is OFF, and when the value is 255, the segment is ON. When the value is in the range from one to 254, the value is a blinking value. In the case of blinking, the ticket with a blink value adds a new ticket with a newly calculated execution time into the timed ticket list. This way the blinking system is continuingly negating the current item value. The calculated time is the segment status multiplied by ten milliseconds.

The ticket with the newly calculated processing time is passed to a function which compares it with the lists first processing value. If it is lower, it is set to be the first item on

the timed list, if not, it is compared with the next one and the timed ticket is added in front of the compared ticket. If the time is lower than the one compared the item is compared with the next item. This loop continues until all the tickets in the list are compared with the new ticket. A removal occurs when the status is changed. The removal function compares the timed ticket's ID and command (which is also known as a ticket type). If they match, the ticket is removed.

5.2.3 Navigation mode

When the user navigates left or right, the control unit is keeping track of the current icon status. The engine's only task in navigation is to the present the changes and to return in the previous state when the navigation mode is closed. The process is handled in the engine by setting the old icon off and the desired icon on. When the navigation is shut the navigation returns into modified original state which is the original but which includes all the state changes (except digit changes) that are pushed from the control unit while the navigation mode is on. The navigation decisions are made at the control unit, and the engine informs the status changes to the control unit by returning the ACK (or NACK) so the control unit can keep track of the icon which the engine is currently blinking (or that the operation has failed). It is decided that the graphics engine is the only instance which keeps track of the on-screen information. This makes the system simpler, but the navigation mode needs to return the currently selected items information. The control unit does not track the position of the currently active item, though it requires the information so it can decide which is the next item that is navigated. The navigation list needs to have the icons ordered in a way that the icon is situated between the next and the previous icons.

When the navigation is set through the FIFO message, the timed ticket list and segment status list are stored in temporary memory spots. When the status is stored, the navigation process can start by setting all the displays off and starting to blink the currently active icon. This depends on the additional information received from the navigation message. The blinking time is pre-set into the YAML configuration files at the set-up.yaml field called "selectblink." The navigation mode is presented in Figure 19.

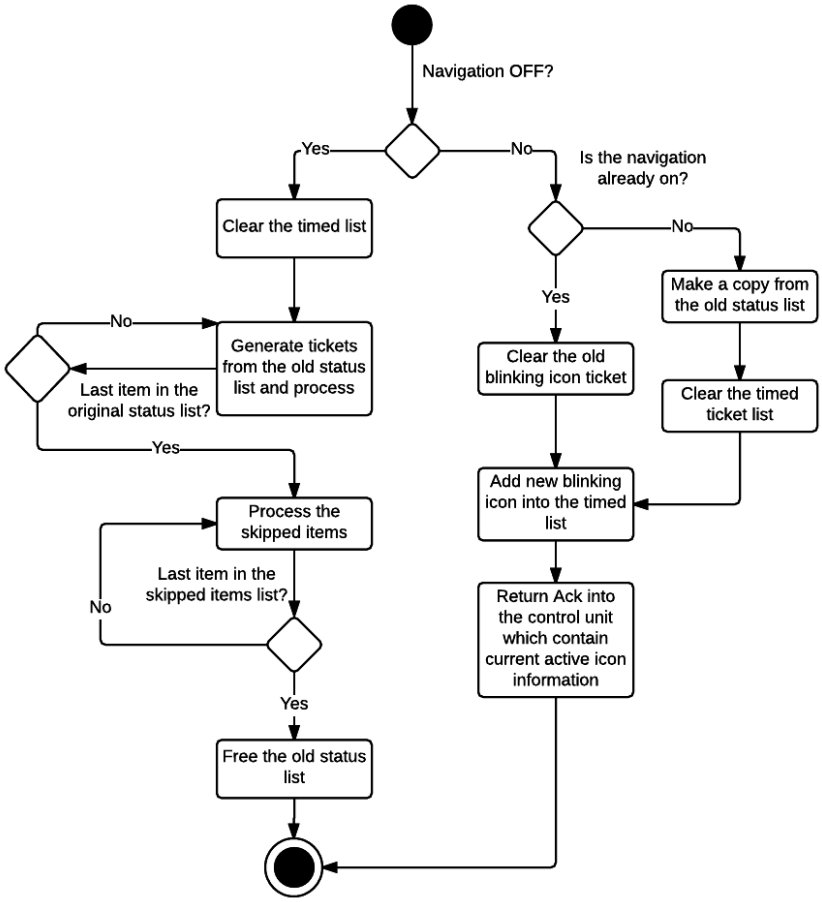


Figure 19: *The navigation process*

When user wants to select the next or the previous screen, the control unit sends screen-navigation message with a display ID specification. To find the correct display, the engine loops the list of icons and compares the first segment's display information to the given display ID. If the icon has the same display ID as the command, the current icon starts to blink and the icon ID is returned to control unit with an ACK message. The display navigation could have been possible to create by making the whole display to blink, but the first icon of the display is chosen because it is better suited according to the previous experience of the University of Tampere. The display navigation could have given the end user more authentic feedback, but as it would require extra selection gesture, it is chosen to be kept simple.

Navigation selecting is done from the control unit which sends the navigation the off message. The navigation mode resets the stored segment and timed lists and then lights the selected icon with the normal icon ON message. The timed list might include tickets that have a lower timestamp than the current time, and all those tickets are processed. They might be tickets that are on the list before the navigation mode, or there might be new ones that are received when the navigation mode is on. The tickets are stored, be-

cause there might be cases about important information which is passed through the engine to the user, for instance errors or faults in the system.

5.2.4 Error handling

The engine sends feedback back into the control unit. The communication back to the control unit is mandatory because the prototype uses the error messages with the navigation managing. However, the two-way communication is not mandatory if the software would not need the navigation system. The error handling would automatically write all the error messages into the log files. The error system is created to help the control unit to know when the operation was unsuccessful (NACK) or successful (ACK). The NACK message is automatically returned when the operations fail, but there is also an option for the controlling program to ask for a ACK message within every command they patch (see Appendix D). The ACK message is used in this project to indicate the control unit which icon is the currently active icon in navigation mode.

The error messages are designed to be as informative as possible. When an error occurs, the error message type reveals the nature of the error. Additional information is attached to the message so that the user can debug the source of the error with ease, for example when an item from the configuration file is not found. The error message can contain specific information about the error. For instance, if user has set an icon to contain three segments but the third one is not found at the configuration files. The error returns indication that the user wanted to do something to the icon, but the system does not have all the necessary segment information set and therefore it cannot find the third segment. The different error types, their returning data values, and the message type is presented in Appendix D.

When the NACK error message is sent, it is also stored into a log file. The data is timestamped, and numeric values are turned into text to ease the reading process. When the engine software is started, the old log file is cleared and a new log is started from an empty file. This is done to prevent the log file from filling the small hard drive. However, when there are errors and the program is started again, the old log file is destroyed and the information is lost. In many cases this is not a matter of concern because the errors are usually regenerating the log file into a same state unless the errors are fixed. The log system needs to be reworked if the product will become a commercial product.

5.3 YAML parser

The YAML parser is a straightforward representation of how the LibYAML works. A single configuration file has a custom parser function, and the user has an ability to print the results. The problem with the parser is that it is in a close relation with the end user. The parser's files are made by humans, and therefore they might contain a large variety of errors, which needs to be resolved. The system can use LibYAML error communication system to provide additional information about the parsing process or where it fails, but the system needs to provide even more information of the configuration data that the LibYAML is not able to test. One of the main security concern is the unsafe functions (STRLEN, CALLOC, and STRCMP). Those functions can be unsafe to use, but the code was considered carefully if the code hold any of those three functions.

The configuration files are transformed into dynamic lists, containing the information which is presented in Figure 16 on page 21. The tests are mostly about testing that the unique IDs are unique and that the loaded values are within boundaries. The most of the values have boundaries which come from the displays or from the internal design, but there are values which are artificially limited to certain values: for instance, the IDs are bound to 10 000 items, which prevents memory errors, and text is limited to 32 characters, which forces the names to be of readable length. One example from the physical point of view is the framerate dimming which is limited into eleven bits by the manufacturer. These boundaries make the system more secure, robust and simpler for the user. Problems arise when these boundaries are needed to change. Then the system must be recompiled.

The parser function works as a separate function so the code runs under every development environment. This allowed the usage of Valgrind, which helped the development process. The system handled a large variety of the user generated errors in the parser function which are stored at the local log files. The problem with this process is that when the software dies, it does not indicate anything back to the user, and the user needs to deduct the failure from the displays or attach a PC and read the log files. When nothing happens on the displays but the lights are on, there is a YAML configuration error. A solution proposal is to run an error sequence inside a display, which would indicate the error to the user.

6. ASSESSEMENT

This chapter is about the assessments of the thesis. The chapter begins with a presentation of how the system is tested and which specific features are tested. The chapter tells also about the testing tools that are used and a custom tool that is created for the user so he or her can control and test the system. The chapter ends with test results from specified test cases.

6.1 Software tests

The tests for this project are: stress and performance tests, white-box testing (peer reviews, path testing), dynamic and static code testing. There is a possibility to do fuzz testing, unit testing and test automation, but there are reasons why they are excluded. The unit testing is excluded because the output is usually pushed straight into the device and there is only one developer who did the implementation for each part of the project. The error messages are logged and the output is at continuous criterion observation and comparison with the logged error messages. The test automation is excluded for the same reasons. The fuzz testing means that the system gets faulty data as messages from the control unit, or an invalid YAML configuration file as an input. This test is removed because the file and message management are not important parts of the project at this moment, and it can be expanded easily if this system is taken into production.

The performance of the project chain from the user input to the displays is measured with the TCP/IP test tool (and CAN recording). The hardware is easy to measure with an oscilloscope, and the software is tested with print functions. The stress testing is done by pushing a great amount of data for a long period to ensure that the system worked correctly after many days. The rest of the tests are more about code quality assurance and bug testing.

6.1.1 Which features are tested

The prototype is created for two purposes: for exhibitions to show the Lumineq display inside a windshield and for the first customers to do simple tests with the glass technology. The prototype is a closed system and thus, the user mistakes, such as configuration file faults, are not so important at this phase of development. The tests focused on the

performance and failure rate. This project is only an exhibition version, and the technologies are not final, and therefore its tests are designed for the demo purpose.

The driver, communication functions and configuration file parsing are simple, and the testing is conducted by peer reviews and through static and dynamic testing. The user mistakes in the parsing can be easily detected from the visual feedback in the glass or from within error messages in the logs. The hardware is problematic, because the SPI line is longer than recommended. It is well known problem within the project, this topic described in more detail at the control unit's master thesis. [3]

The focus in testing is the command managing. The command managing system is spread into multiple branches which handle multiple dynamic lists and which have many features with a complex structure and a small number of lines of code. The system has strict rules which are necessary. If some part is failing, the whole system does not work at all. However, the branching and strict rules make the system easy to debug and the code more robust. The test methods here are stress test, performance test, peer review, and predetermined test cases.

6.1.2 Test tools

Most of the tools are basically de facto standard for the C programming with Linux. GDB is the main debugger, Valgrind tests for memory leaks and oscilloscope is used for measuring and debugging the hardware and the system performance. Static test software called Splint is also used, but the most of the Splint's test results should be ignored because the software cannot recognise the code correctly in multiple cases. A good example of this is a symbolic variable defining where the readability and code quality improves, but Splint finds the case incorrect and passes a warning.

One task is to create a simple TCP/IP tool that has three different types of usage: controlling, testing, and automation. The main screen of the tool is presented in Figure 20. The TCP/IP tool is created with C++ and QT. Initially, the test software is designed for the developers' personal usage, but in the middle of the project a similar system was requested by the client. The software is modified for the new purpose by adding SSH communication to transfer files between the target machine and PC, adding an ability to reboot the software or the whole device from the PC. The software is used to demos for exhibitions. Including the demos into the test software was a mistake, because the device requires a PC for the connection, and it revealed to be more time consuming than creating an animation system inside the device.

The test software's architecture design was faulty from the beginning. The C++ and QT are not favourable tools to create a small tool like this, and the changes in the middle of

the project broke the system. The system should have been written again from the scratch and not to expand the software. The problems that occurred at a later project are hard to manage, and it hampered the prototype's development because there are multiple cases where it took a long time to figure the sources of errors, both in the prototype or the test software.

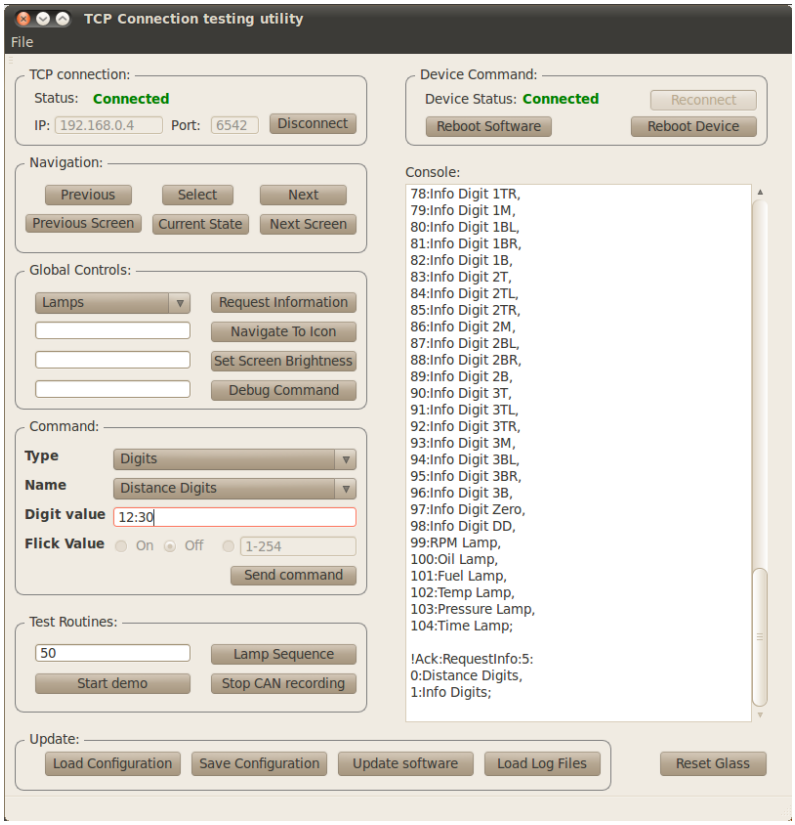


Figure 20: *The control / test software*

The custom test software controls the displays via the Linux software. The SSH connection is used to download log files and current configuration files from the Linux machine. The software can also update the software and send new configuration files to the machine. The SSH is also used to reboot the software and the device itself through remote access. The device can be used to start or stop the CAN recording, and it has two built-in demos and a sequence loop that loops through all segments twice, first setting everything on, and on the second run closing the segments. Its main purpose is to push updates onto the displays and test the new configuration files. The system loads the configuration information from the device as a plain text (the unique IDs of the items), which eases the finding of the desired changes for the user. The software also contains a console that prints all the returned messages from the control unit.

6.2 Test results

The test pass criteria measure the message speed that are passing through the system, bug free chain of events, and that the displays update smoothly for the user. The minimum update time for a single packet is slowed to ten milliseconds for a better overall quality for the user. This is the maximum time that the updates have for passing the update chain to the displays. It functions also as the system's sleep time. The maximum time affects the test passing criteria.

The speed results were tested with three different test cases. The first test case is for the driver performance. The test measured the time from the engine's IOCTL write to the device drivers SPI send with a single packet. The IOCTL send time is printed and it is compared to the SPI transfer's start time obtained via an oscilloscope. The second test examined the time that a single packet takes from the driver's SPI send function into the update sequence of the display. This time is measured solely with the oscilloscope. The third test concerned the maximum time from TCP test software to the display update sequence. This test used digit command so the engine passes 23 packets which were transformed into three display packets. The test results are presented in Figure 21.

	Driver performance	Driver send to display update	Digit chain update test
Maximum allowed:	3 milliseconds	7 milliseconds	30 milliseconds
Results:	880 microseconds	2 milliseconds	26 milliseconds

Figure 21: *The software project overview*

The speed test was a success and it passed the software's passing criteria. The last two results would be faster, if the kernel's SPI interface supports the GPIO pins natively. Now when the GPIO functionality is done by register changes, the system does not reach its full potential. The third test result could also be faster, if the lamp information could be edited straight inside the graphical engine. This can be achieved by changing into a newer version of Linux kernel and by replacing the current driver with SPIDEV.

The second main test criteria are about the messages that are passed to the engine. The messages must be pushed to the screen with a full hundred percent percentage, unless

there is an error about the access rights, or if the update is already on the screen. This is tested by building up a test that should try to make changes that were not allowed. The test result is examined from two sources. The first and more important one is how the displays reacted to the human eyes. The second source is the error logs that are examined. Both the error logs and the examination results passed the criteria.

The third passing criteria is that the system needs to run without crashing for certain amount of time. Because this system is initially designed for an exhibition demo, the system needed to run only a day without any errors. The system did run for an even longer period. This pass criteria needs to be changed into “never fails” before this product is taken into the production.

There were also a few “happy accidents” where the demo glass broke, as shown in Figure 22. The breakdown showed the team that the displays can work even if the glass breaks. This is useful information considering the future. When the product is taken into the production stage, the glass needs to work even if the windshield slightly breaks. This accident proves that the technology works in real environments.

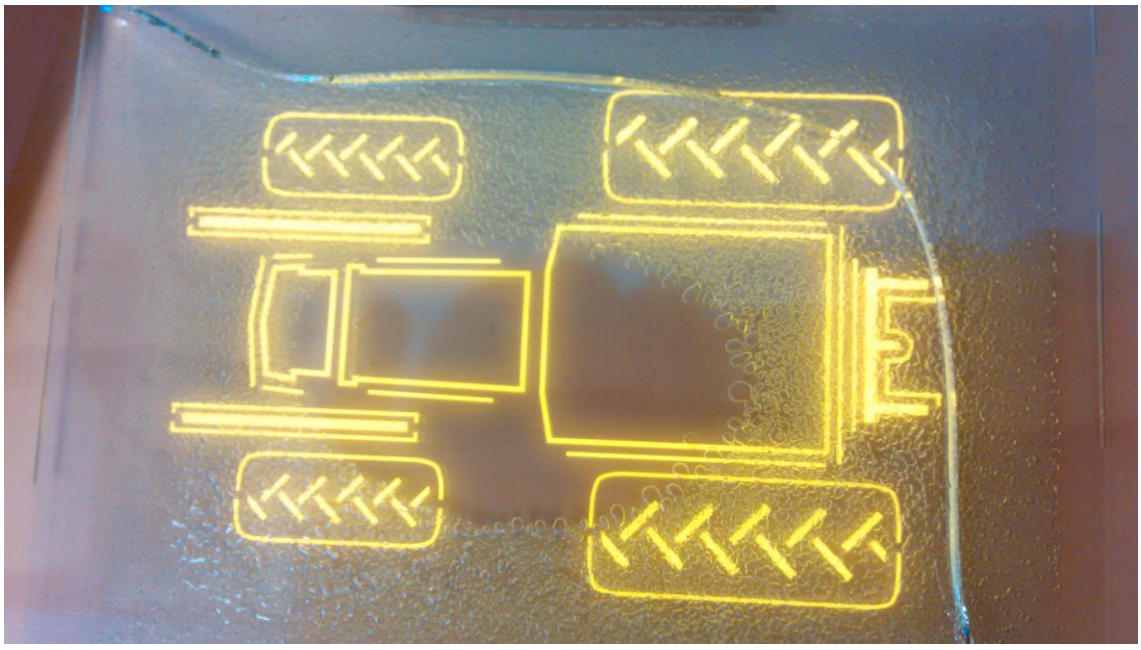


Figure 22: *The display technology works even when the windshield is broken*

6.3 Future features

The main problem with the whole prototype is the display control board’s SPI bus which is not the best solution for the product. The SPI line affects many aspects in the

design and the architecture of the system. Changing the SPI for a CAN bus would remove the need for extra electronics, and therefore the need for the drive module. The CAN also affects the target device. It can be changed to a one with more modern Linux kernel, and this way it enhances the security of the system. The change to CAN also affects the send procedure of the engine. IOCTL calls need to be removed and changed into CAN messages which can be used from the kernel's own CAN library. The only problem is that the new send function must support all the CAN protocols which are used in the vehicles where the product is used. There is also a possibility to have a wireless connection if the display control units supported the same technologies. This will make the system even simpler by removing the need for CAN output.

There are features which were originally planned, or was proposed in the middle of the development process. The main idea was to add an ability to make animations inside the YAML configuration files. This feature was not necessary for the prototype and it was excluded from the development backlog. At the end of the project this was a mistake because the demonstration animation does need to run in a separate computer which is not suitable for the demo purposes. There was an idea that a YAML setup file would hold a set of items which would be on when the system starts. This is especially important when the displays start to have different contents. The navigation mode must be reconsidered. The navigation is now a part of the prototype to demonstrate University of Tampere's haptic feedback, but if it is not used for the target vehicles, it would be more usable and safer to use the current display system only as an informative display.

On the technical side, there are features which should be added to the prototype to make it more robust. The first set of features is about the navigation mode (if it is planned to be still included in the system). The current version of the navigation system does not have any timeout. If an error occurs within the navigation tools while navigating, the system should return to its original state after a certain time, so the displays will give at least some information if the other devices crash. The ticket system runs normally in the background when the navigation is set on, except for the digit tickets which are automatically deleted. They are loaded onto a dynamic list, but the problem is that if there is a great number of tickets which are all passed into the memory, then the memory is filled. It is better to make a ring buffer which would have a fixed size, and when reaching its limit, it would start overwriting the first tickets.

Another resource error might occur when the log files become too large. There is no limit to how large the log files can get. When the system runs for a long period, the log files can form a problem. When an error occurs, it might happen in a situation where it is a part of a larger updating process, for example digit value display action. It will run the update but it will be missing parts of an update where the error occurs. A good way to disable this is to backtrack the last edit and undo all the actions which are done with the driver's edit. This is not a large problem, because the system returns NACK into the control unit which is updated to resend the command on some occasions. There are a set

of error messages that are returned to the user. Now, it is limited to few obvious cases but there is a possibility to expand them even further with more by adding more accurate feedback to the user. The YAML parser has a good error reporting system, but currently the engine produces only simple error messages.

The software is tested for the team’s needs. When the system proceeds into production, it requires more testing and a further security checkout. The product can take an advantage from multiple different test methods. The main testing methods are unit tests and fuzz testing, but the largest effort should be put into test automation. The automation can be inside the software but also because the screens are static, the feedback can be tested with cameras which take a set of pictures from the user feedback and analyse them. When the software is pushed into production, the software requires a security check. The main security enhancements are inside the device itself, but the main problem with the software is the user input and the YAML file processing. Additionally, there might be logical problems that do not affect the software but the user.

7. CONCLUSIONS

The task for this thesis was to create a graphics display engine for a windshield with electroluminescent displays. The system was designed for exhibition use which affected the variety of features. The system was developed the future use purposes in mind, and the system was also designed to be a basis for the production software. The main research focuses were the architectural solution and the study of the technology, how it works in real situations. The system was embedded with additional features which expand the few simple features provided by the display technology. The thesis also took a stand on how the major features were developed.

The project was successful. The software's architecture supports a large variation of devices and will work as a basis for the production phase. All the planned features were not included because the focus was placed on the basic features and the overall quality of the code. There were no known bugs, and the all the tests passed the testing criteria. The performance was also in the predetermined limits.

The studies of the current technology brought additional information about the current system for the company, which helped the company to understand and respond to the theoretical and practical problems and limitations of the hardware and the chosen technologies. There were multiple open questions which were reported to the client. These problems are mostly about the display's physical features that were problematic when concerning the target environment. The largest problem was the SPI lines' physical limitations which make some random errors pop up in the displays. The limitations' affects were minimized as far as possible.

The project architecture was designed for the future demand. The main idea was to create a split architecture which has multiple processes that can be developed separately. This way the technologies can change through the development, and depending on the used setup, there are pieces of software for different technologies which communicate together: for instance, if the display's technology radically changes, only the graphical engine needs to be rewritten. The current architecture also allows even other systems (for example haptic feedback) to be connected to the current software.

Through the development, the system was developed for an end user and the future developers in mind. This way the usability and feedback features were moulding the software. The main concern was to make the system as simple as possible, and the project succeeded in that account. The system is fully automated and the system assembly consists of only attaching a few cords through the system. This design is also supported by

the electronics and the cover design. The custom test software was also created to support this design pattern so all the updates can be passed to the device with that software, even by non-technical persons.

SOURCES

[1] J. Corbet, A Rubini, G. Kroah-Hartman, Linux Device Drivers 3rd edition, 2005, pages 1-3, 137-139, [referred 28 February 2016]. Available at: http://free-electrons.com/doc/books/ldd3.pdf

[2] Displays for extreme conditions, Beneq Products Oy, [referred 28 February 2016]. Available at: http://lumineq.com/sites/default/files/documents/lumineq_tfel_brochure.pdf

[3] T. Rekosuo, Näyttölasin kontrolliyksikkö, Tampereen Teknillinen Yliopisto, 2015. Available at: http://dspace.cc.tut.fi/dpub/bitstream/handle/123456789/23503/rekosuo.pdf

[4] Serial Peripheral Interface (SPI), SparkFun Electronics Inc, [referred 28 February 2016]. Available at: https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi

[5] K. Simonov, LibYAML, 2014, [referred 28 February 2016]. Available at http://pyyaml.org/wiki/LibYAML

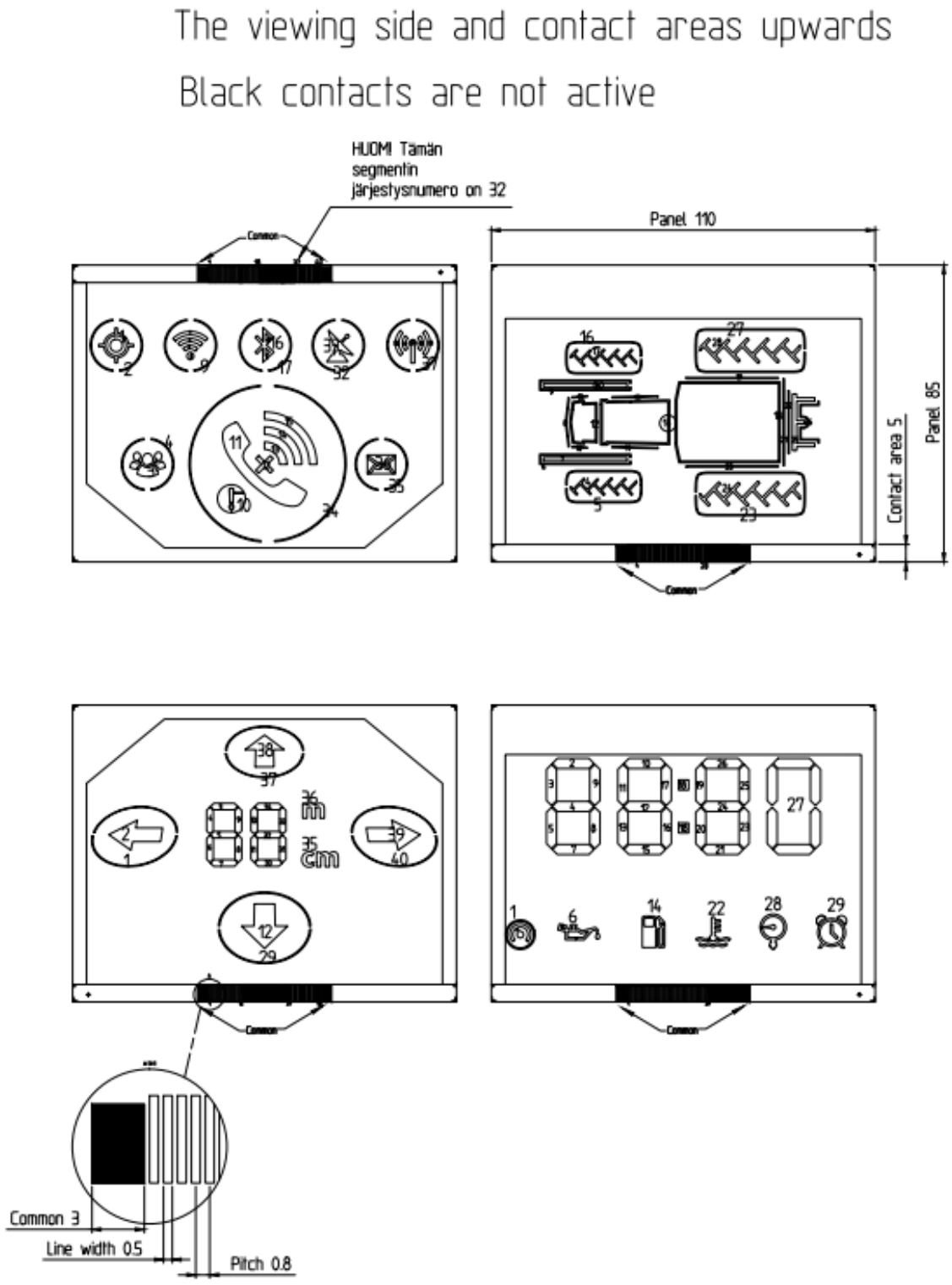
[6] SPI Block Guide, Motorola Inc., 2001, p 26-29. Available at: https://web.archive.org/web/20150413003534/http://www.ee.nmt.edu/~teare/ee308l/datasheets/S12SPIV3.pdf

[7] Thin film electroluminescent displays, Beneq Products Oy, [referred 28 February 2016]. Available at: http://lumineq.com/en/technology

[8] YAML about, YAML.org, [referred 28 February 2016]. Available at: http://www.yaml.org/about.html

[9] YAML Ain't Markup Language (YAML™) Version 1.2, 3rd edition, 2009. Available at: http://www.yaml.org/spec/1.2/spec.html

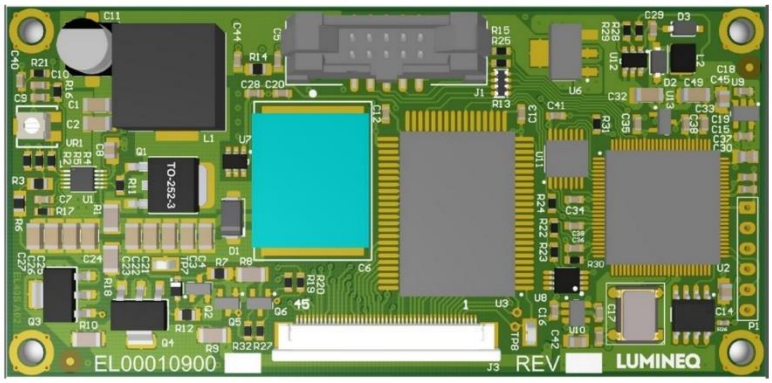
APPENDIX A: SEGMENT MODEL



APPENDIX B: EXAMPLE OF LAMP CONFIGURATION FILE

```
# Configuration file for Middleware
#       This document handles all lamp controlling configurable values from the
#       middleware software.
#
#       This file has the information about individual lamps in screens
#
#       Use 8 spaces, tabs are not allowed!
#
#       Version 1.0
#       Miikka Juomoja & Tommi Rekosuo, Tampere University of Technology

---
#LAMPS
# Notice that with this revision, the 1st screens lampid's are reversed!

#  .. :: Example :: ..
#- lname:     GPS Icon      # Unique name of the lamp
#  scrid:     1             # Screen ID where the lamp is located
#  lmpid:     40            # Physical lamp number

# Communications screen
# GPS
- lname:     GPS Icon
  scrid:     1
  lmpid:     40

- lname:     GPS Circle
  scrid:     1
  lmpid:     39

# WLAN
- lname:     WLAN Ball
  scrid:     1
  lmpid:     33

- lname:     WLAN Inner Arc
  scrid:     1
  lmpid:     36

- lname:     WLAN Middle Arc
  scrid:     1
  lmpid:     35

- lname:     WLAN Outer Arc
  scrid:     1
  lmpid:     34
...
```

APPENDIX C: EL40S ELECTRONICS OPERATION MANUAL



EL40S Electronics

Operation Manual

Beneq Products Oy
Olarinluoma 9
FI-02200 Espoo
Finland

Tel. +358 9 7599 530
Fax +358 9 7599 5310
lumineq@beneq.com

VAT ID FI25115461
www.beneq.com
www.lumineq.com

Date: March 3, 2015

Document number: ED000894 DRAFT

Table of contents:

1 Overview

The EL40S electronics is designed for demonstration purposes with the Lumineq TASEL glass.

1.1 Features

- 4-bit SPI Interface
- Locking connector input connector
- FPC connector or flex cable connection for the TASEL glass

2 Installation and handling

The EL40S uses CMOS and power MOS-FET devices. These components are electrostatic-sensitive. Unpack, assemble, and examine this assembly in a static-controlled area only. When shipping, use packing materials designed for protection of electrostatic-sensitive components.

WARNING: These products generate voltages capable of causing personal injury (high voltage up to 235 VAC). Do not touch the display electronics during operation

2.1 Cable length

A maximum cable length of 0.6 m (24 in.) is recommended. Longer cables may cause data transfer problems between the data transmitted and the display input connector. Excessive cable lengths can pick up unwanted EMI.

3 Specifications

Performance characteristics are guaranteed when measured at 25 °C with rated input voltage unless otherwise specified. The minimum and maximum specifications in this manual should be met, without exception, to ensure the long-term reliability of the display. Beneq Products does not recommend operation of the display outside these specifications.

CAUTION: Absolute maximum ratings are those values beyond which damage to the device may occur.

3.1 Display Input Connector

The display uses the Samtec EHT-105-01-1-D-SM-LC or equivalent locking connector. The mating connector is in the Samtec TCSD family of cable strips. The proper connector, user-specified cable length and connector configuration is supplied as a single unit. Consult your Samtec representative for the cable/connector options. Compatibility with non-Samtec equivalents should be verified before use. The part number of Samtec mating connector without a cable is the TCSD-05-01-N.

Table 1. SPI pin assignments

Symbol	Pin	Pin	Symbol
+12 VDC	1	2	GND
SCLK	3	4	GND
MOSI	5	6	GND
SS	7	8	GND
ST	9	10	Reserved

Table 2. SPI input description

Signal	Functional Description
+12 VDC	Display power supply
GND	Display ground
SCLK	Clock from Master
MOSI	Master Output Slave Input
SS	Slave Select
ST	Self-Test
Reserved	Reserved for Master Input Slave Output (MOSI), Do not connect

3.2 Display Input Signal Levels

Table 3. Display input signal levels

	Minimum	Maximum	Units
Absolute Maximum input Voltage	- 0.5	4.1	V
Low level input voltage (Nominal 0 V)	0	0.8	V
High level input voltage (Nominal 3.3 V)	2.0	3.8	V

3.3 TASEL Glass Connections

Table 3. TASEL Glass connector

Pin	Description
1	Common
2	Common
3	Not connected
4	Segment 1
|	|
43	Segment 40
44	Not connected
45	Common

Connector: FPC (Flexible Printed Circuit) Connector, 0.5 mm pitch, 45P, Gold Contact Plating, Bottom contact, 0.5 A, 250 VAC, Manufacture: TE Connectivity, Part number: 4-1734592-5.

Table 4. TASEL Glass Flex contact pads

PIN	Description
41	Common
1	Segment 1
|	|
40	Segment 40
42	Common

Flex PCB connection: 40 segments, 2 commons (One common each side of flex)
Flex pitch: 0.8 mm, total 40.3 mm (40 x 0.5 mm pad, 41 x 0.3 mm gap, 2 x 3 mm common, 2 x 1 mm side)

3.4 SPI Video Timing

The SPI interface is driven with the rising edge of SCL. A falling edge on LD signal indicates the beginning of an access on the SPI Interface, the rising edge of LD determines an access on SPI. An access must consist of exactly 16 bits for write operation. The timing restrictions on SPI are defined in below figure 1 and table:

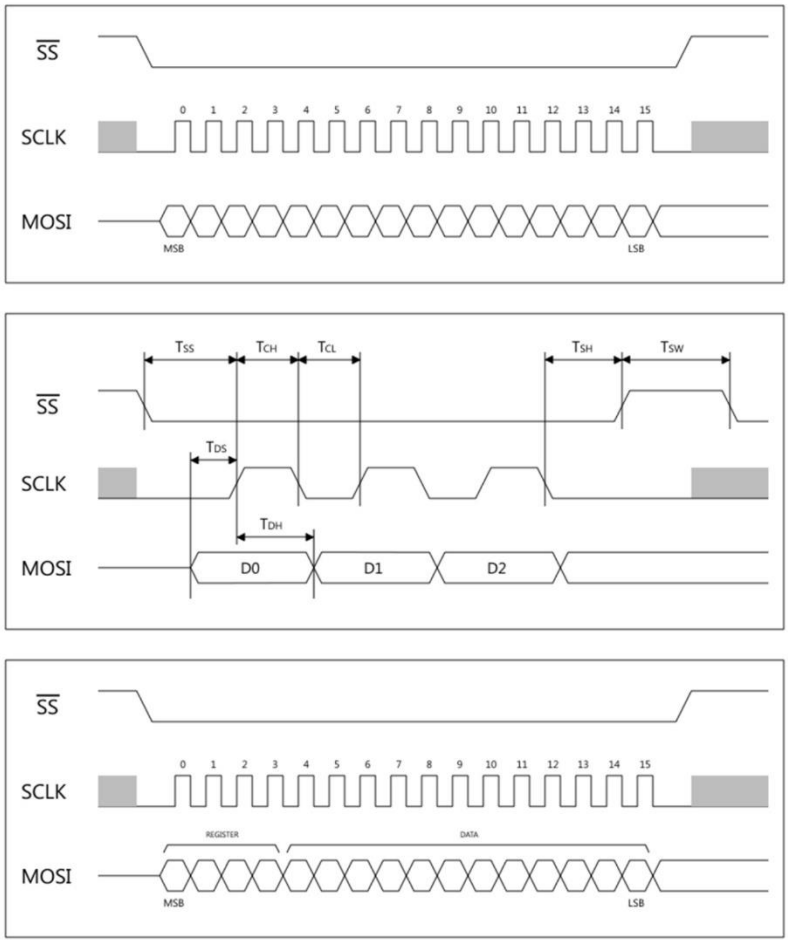


Figure 1. SPI video timing

Table 3. SPI timing restrictions

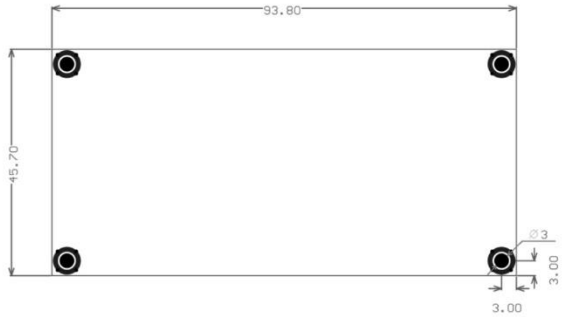
		Min (ns)
SCLK high time	T_{CH}	150
SCLK low time	T_{CL}	150
SS -> SCLK setup time	T_{SS}	150
SCLK -> SS hold time	T_{SH}	150
SS disabled between cycles	T_{SW}	150
Data setup time	T_{DS}	150
Data hold time	T_{DH}	150

3.5 Self-test

Under Development /to be tested: The display input connector has a self-test input. When this input is continuously at low state, the display enters to the self-test mode. At this mode all segments are turned on, and the display brightness is set to full.

3.6 Component board mechanical dimensions

Dimensions: 93.8 x 45.7 mm
Height: 14 mm
Component envelope height: 15 mm



Picture 1. PCB Dimensions

Beneq Products reserves the right to relocate components within the constraints of the component envelope without prior customer notification. For this reason, Beneq Products advises users to design enclosure components to be outside the component envelope.
Device designers will need to consider their specific system requirements to determine the spacing necessary to maintain the specified ambient temperature.
Air flow and surrounding component materials will affect the depth of the air gap.

3.7 DC input requirements

All internal high voltages are generated from the display supply voltage. The supply voltage should be present whenever video input signals are applied.

There is no overcurrent protection on either the supply voltage input to protect against catastrophic faults. Beneq Products recommends the use of a series fuse on the 12 volt supply. A general guideline is to rate the fuse at 1.8 to 2 times the display maximum current rating.

Table 4. DC Input Voltage Requirements

Description	Min.	Max.	Units
Supply voltage (Nominal = 12 VDC)	10.8	13.2	V

Table 5. DC Input Current Requirements

Description	Current	Units
Typical input current, (Supply voltage = 10.8 VDC) (1)	0.5	A
Typical input current, (Supply voltage = 13.2 VDC) (1)	0.4	A

Table 6. DC Display Power Requirements

Description	Power	Units
Display typical maximum power (1)	4.7	W
Display typical minimum power (2)	0.7	W

Notes (tables 5 and 6):

1. These values are display glass specific, which mainly depends on the TASEL glass size and brightness. The typical values are for the display glass with 1000 mm^2 illuminating area and 350 cd/m^2 brightness.
2. All pixels off
3. Notes: Maximum DC input current mainly depends on the TASEL glass size and brightness.

3.8 Estimated power budget calculation

A rough estimated power budget can be calculated as follows:

Typical electronics efficacy: 11.4 W/cd (TBD)
Typical display electronics typical minimum power = 0.7 W
Brightness= A cd/mm^2
Active area= B m2

P (W) = 11.4 W/cd * A cd/m^2 * B mm^2 /10^6+ 0.7 W

3.9 Environmental

Table 1. Environmental characteristics

Temperature	
Operating	TBD
Operating Survival	TBD
Storage	TBD
Humidity	
Non-condensing, operating	TBD
Condensing, non-operating	TBD
Altitude	
Operating/non-operating	TBD
Vibration	
Random Operating/non-operating	TBD TBD
Shock	
Operating/non-operating	TBD

3.10 Reliability

TBD.

3.11 Safety and EMI performance

TBD.

3.12 Optional features

Conformal coating is available as an option.

4 16 bit SPI Interface Protocol

4.1 16-bit Serial Data Format

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Register address				Data											

4.2 Register Address

Register	D15	D14	D13	D12	Notes
Segment 0	0	0	0	0	Segments 1-12
Segment 1	0	0	0	1	Segments 13-24
Segment 2	0	0	1	0	Segments 25-36
Segment 3	0	0	1	1	Segments 37-40
Frame rate	1	0	0	1	Dimming
Voltage	1	0	1	0	Dimming

4.3 Register Segments

Register	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Segment 0	12	11	10	9	8	7	6	5	4	3	2	1
Segment 1	24	23	22	21	20	19	18	17	16	15	14	13
Segment 2	36	35	34	33	32	31	30	29	28	27	26	25
Segment 3	-	-	-	-	-	-	-	-	40	39	38	37

Segment register value 1 = Segment ON, value = segment OFF

Note: Refer to custom TASEL drawing for segment numbering.

4.4 Frame rate dimming

Frame rate	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	Notes
Minimum	0	0	0	0	0	0	0	0	0	0	0	0	70 Hz (1)
|	|	|	|	|	|	|	|	|	|	|	|	|	|
Maximum	1	1	1	1	1	1	1	1	1	1	1	1	0.5 -1 kHz (2)

Notes:
1. > 60 Hz
2. Currently 500 Hz / Under development

4.5 Voltage Dimming

Voltage	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	Notes
Minimum	-	-	-	-	0	0	0	0	0	0	0	0	0 cd/m2
|	|	|	|	|	|	|	|	|	|	|	|	|	|
Maximum	-	-	-	-	1	1	1	1	1	1	1	1	Max

Note: Under development

5 Description of warranty

Seller warrants that the Goods will conform to published specifications and be free from defects in material during warranty time from delivery. To the extent that goods incorporate third-party-owned software, seller shall pass on seller's licensor's warranty to buyer subject to the terms and conditions of seller's license.

Warranty repairs shall be warranted for the remainder of the original warranty period. Buyer shall report defect claims in writing to seller immediately upon discovery, and in any event, within the warranty period. Buyer must return goods to seller within 30 days of seller's receipt of a warranty claim notice and only after receiving seller's return goods authorization. Seller shall, at its sole option, repair or replace the goods.

If goods were repaired, altered or modified by persons other than seller, this warranty is void. Conditions resulting from normal wear and tear and buyer's failure to properly store, install, operate, handle or maintain the goods are not within this warranty. Repair or replacement of goods is seller's sole obligation and buyer's exclusive remedy for all claims of defects. If that remedy is adjudicated insufficient, Seller shall refund buyer's paid price for the goods and have no other liability to buyer.

All warranty repairs must be performed at seller's authorized service center using parts approved by seller. Buyer shall pay costs of sending goods to seller on a warranty claim and seller shall pay costs of returning goods to buyer. The turnaround time on repairs will usually be 30 working days or less. Seller accepts no added liability for additional days for repair or replacement.

If seller offers technical support relating to the goods, such support shall neither modify the warranty nor create an obligation of seller. Buyer is not relying on seller's skill or judgment to select goods for buyer's purposes. Seller's software, if included with goods, is sold as is, and this warranty is inapplicable to such software.

SELLER DISCLAIMS ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

6 Ordering information

Product	Part Number	Description
ECA, EL40S	EL00010900	Electronics board for the TASEL demonstration purposes. SPI interface.
ECA, EL40s w FPC	EL00010901	Electronics board for the TASEL demonstration purposes. SPI interface, Locking FPC connector.

Design and specifications are subject to change without notice.

Beneq continues to provide optional, and in many cases custom, features to address the specific customer requirements. Consult Beneq Sales for pricing, lead time and minimum quantity requirements.

7 Support and service

Beneq Products is a Finnish company based in Espoo, Finland, with a world-wide sales distribution network. Full application engineering support and service are available to make the integration of Lumineq displays as simple and quick as possible for our customers.
RMA Procedure: For a Returned Material Authorization number, please contact Beneq Products Oy by email (rma.lumineq@beneq.com) with the model number(s), serial number(s) and brief description of the problem. When returning goods for repair, please include a brief description of the problem, and mark the outside of the shipping container with the RMA number.

8 RoHS II

Beneq Products Oy is committed to continuous improvement. As part of this process we are fully in support of EU directive 2011/65/EU, the Restriction of Hazardous Substances, commonly known as RoHS II or RoHS Recast, which, compared to RoHS, keeps the restrictions on the original six hazardous substances, including lead (Pb) in electronic equipment. It also expands these restrictions to previously exempted categories including medical devices and monitoring and control instruments.

APPENDIX D: INTERNAL MESSAGES DOCUMENTATION

```
0-------------------------------0
|      OVERALL INFORMATION      |
0-------------------------------0
```

This document is about the internal messaging system inside the two Middleware processes.

VERSION	EDITOR	DATE	CHANGES
0.1	MJ	5.1.15	The first version of the documentation.
0.2	MJ	16.1.15	Corrected the message data, added voltage dimming and removed dimming at lamp control
0.3	MJ	21.1.15	Added Screen control, lamp/ group blinking, digit data and unglighting segments
0.3	TR	6.2.15	Added navigation mode command, added details of each command
0.4	MJ	12.2.15	Edited digit command and icon command
0.5	MJ	2.3.15	Removed Unlight all cmd (do with it screen cmd)|
0.6	MJ	8.5.15	Added return message and added success
1.0	MJ	1.6.15	Removed animation and added it to free for use

Table of Contents:


```
0-------------------------------------------------------------------------------0
```

```
0-------------------------------0
|         1.1 OVERVIEW          |
0-------------------------------0
```

The messaging system base is presented here. All the messages are in little-endian format.

Command	ID [2 bytes]	Succ	Info
1 Byte		1 Byte	1 Byte

Time = Time measured in scale of 0.1 seconds.

Command = Command byte which tells the program what to do:
- 0x00 = Free for use
- 0x01 = Screen control
- 0x02 = Icon control
- 0x03 = Lamp control
- 0x04 = Group control
- 0x05 = Digit
- 0x06 = Set maximum dimming (Hz)
- 0x07 = Set maximum dimming (Voltage)
- 0x08 = Navigation mode
- 0x09 = Return message (ONLY SPI --> CAN)
- 0x0A = Debug light

ID = Informational data:
- 0x00 = Free for use
- 0x01 = Screen ID (CS)
- 0x02 = Icon ID
- 0x03 = Lamp ID
- 0x04 = Group ID
- 0x05 = Data
- 0x06 = Framerate dimming value
- 0x07 = Voltage dimming value
- 0x08 = ID
- 0x09 = Type of error/ Type of return
- 0x0A = HIGH BYTE = Screen, LOW lamp ID on screen

Success = if 1, then the successful processing needs to return ACK
(THIS IS ONLY FOR MESSAGES FROM CAN --> SPI)

Info = Additional information:
- 0x00 = Free for use
- 0x01 = 0 = kill, 255 = start, 1-254 blink 0.1s
- 0x02 = 0 = kill, 255 = start, 1-254 blink 0.1s
- 0x03 = 0 = kill, 255 = start, 1-254 blink 0.1s
- 0x04 = 0 = kill, 255 = start, 1-254 blink 0.1s
- 0x05 = Digit ID
- 0x06 = Screen ID (CS)
- 0x07 = Screen ID (CS)
- 0x08 = Type 7 bits | on/off 1 bit
- 0x09 = Data numbering (See below)
- 0x0A = Value

2.1 COMMAND DETAILS

Command	ID	Info
0x00 Free for use		
0x01 Screen control	Screen ID	0 off, 255 on, 1-254 blink rate in 0.1s
0x02 Group control	Group ID	0 off, 255 on, 1-254 blink rate in 0.1s

0x03 Lamp control	Lamp ID	0 off, 255 on, 1-254 blink rate in 0.1s
0x04 Digit	1st bit ddot1 2nd bit ddot2 Last 14 bits = value	Digit ID
0x05 Set Maximum dimming	Dimming value	Screen ID (CS)
0x06 Set Maximum dimming	Dimming value	Screen ID (CS)
0x07 Unlight all segments	Screen ID (0 = all)	NOT USED
0x08 Navigation mode	ID	Type 7 bits 0 Icon 1 Screen On/off 1 bit 0 Off 1 On
0x09 Return	Type (see below)	Dynamic data (see below)

4.1 RETURN MESSAGES

These messages are the return messages from the SPI to CAN so the CAN code can return information into the UI.

CMD = Always the same (see before)
ERR = Return message type (see below)

Return message types:

0. SUCCESS/ UNSUCCESS
1. NAVIGATION MESSAGE (current screen and icon)
2. FIFO FAILURE (from CAN to SPI)
3. NO ACCESS (priority is lower than the reserved)
4. DRIVER SEND FAILED
5. ITEM NOT FOUND (Basically YAML error)

0 SUCCESS/ UNSUCCESS:

CMD	ERR	ISSUCC	TYPE	ID	DATA
1 Byte	[2 bytes]	1 Byte	[2 bytes]	[2 bytes]	[2 bytes]

ISSUCC = 0 if successful
1 if failure
TYPE = Ticket type (lamp, group, icon, ...)
ID = Ticket ID
DATA = Ticket's data value

1 NAVIGATION MESSAGE:

CMD	ERR	SCREEN	ICON
1 Byte	[2 bytes]	[2 bytes]	[2 bytes]

SCREEN = In which screen the navigation is currently (ID)
ICON = In which icon the navigation is currently (ID)

2 FIFO FAILURE:

CMD	ERR
1 Byte	[2 bytes]

3 NO ACCESS:

CMD	ERR	TYPE	ID	TYPE	ID
1 Byte	[2 bytes]	[2 bytes]	[2 bytes]	[2 bytes]	[2 bytes]

TYPE 1 = Ticket type (lamp, group, icon, ...) from the CAN
ID 1 = Ticket ID from the CAN
TYPE 2 = Ticket type (lamp, group, icon, ...) which has the reserve
ID 2 = Ticket ID which has the larger priority reserve

4 DRIVER SEND FAILED:

CMD	ERR
1 Byte	[2 bytes]

5 ITEM NOT FOUND:

CMD	ERR	TYPE	ID	TYPE	ID
1 Byte	[2 bytes]	[2 bytes]	[2 bytes]	[2 bytes]	[2 bytes]

TYPE 1 = Ticket type (lamp, group, icon, ...) from the CAN
ID 1 = Ticket ID from the CAN
TYPE 2 = Ticket type (lamp, group, icon, ...) which was not found
ID 2 = Ticket ID which was not found

0----------0

5.1 OTHER VARIANCES

This chapter is about the all the variances from the base, what has changed and
where it is used.

..::Ticket::..

TIME	CMD	ID	SUCC	INFO
[4 Bytes]	1 Byte	[2 bytes]	1 Byte	1 Byte

In ticket system there are two major changes. First the messages are changed into structs. The second change is that the time value is added to the start.

0----------0